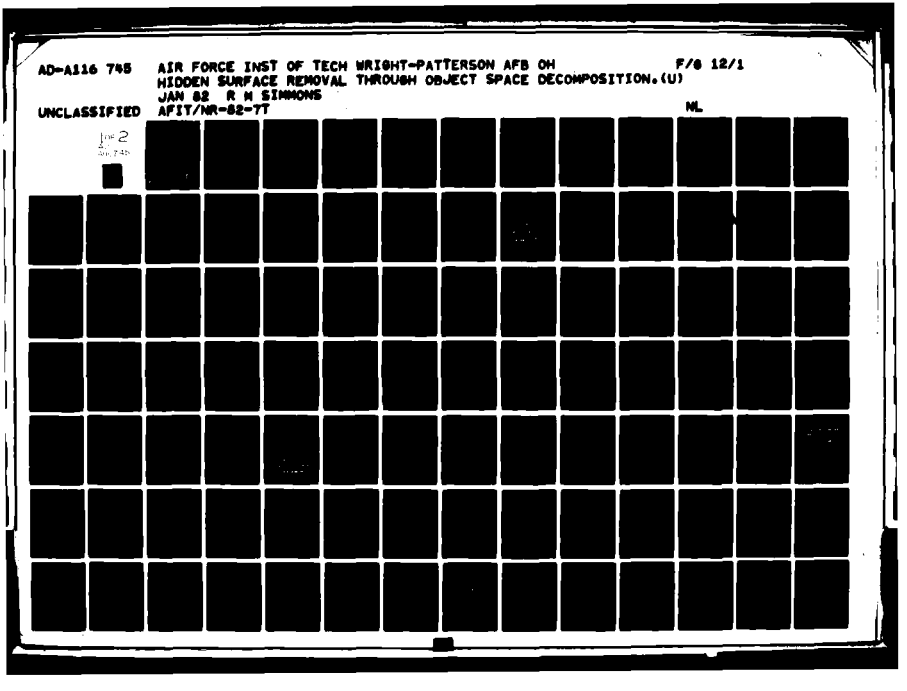

AD-A116 745
AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
F/G 12/1
HIDDEN SURFACE REMOVAL THROUGH OBJECT SPACE DECOMPOSITION.(U)
JAN 82 R H SIMMONS
UNCLASSIFIED
AFIT/NR-82-7T
ML

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFIT/NR/82-7T	2. GOVT ACCESSION NO. A116745 / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Hidden Surface Removal Through Object Space Decomposition	5. TYPE OF REPORT & PERIOD COVERED: THESIS/DISSERTATION
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Robert M. Simmons	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: AFIT STUDENT AT: Massachusetts Institute of Technology	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: AFIT/NR WPAFB OH 45433	12. REPORT DATE: Jan 1982
	13. NUMBER OF PAGES: 128
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASS
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

LYNN E. WOLAVER
Dean for Research and
Professional Development
AIR FORCE INSTITUTE OF TECHNOLOGY (ATC)
WRIGHT-PATTERSON AFB, OH 45433

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-17
22 JUN 1982

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

82 07 07 065

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

HIDDEN SURFACE REMOVAL

THROUGH OBJECT SPACE DECOMPOSITION

by

ROBERT MONROE SIMMONS

Submitted to the Department of Electrical Engineering and Computer Science
on January 14, 1982 in partial fulfillment of the requirements for
the Degree of Master of Science

ABSTRACT

Hidden surface removal is a computer graphics problem involving a great deal of computation. The problem involves two facets: determining which objects should appear in front of others (prioritization), and elimination of invisible portions of the objects through geometric calculations. Prioritization is accomplished using object space decomposition, which divides object space in a binary fashion such that the objects in a scene (or critical portions of those objects) occupy unique sub-volumes of the object space. An octal-tree is used to map the decomposition, and a simple traversal of the tree, with minor interruptions for more sophisticated decision-making, results in a stream of objects in priority order.

The second phase of the hidden surface problem, removal of invisible portions of objects, often requires a great deal of computation. Parallel processing offers potential for savings in response time, and the second part of this thesis investigates a number of algorithms which attempt to take advantage of inherent concurrency. Three algorithms are presented: a quad-tree image space decomposition algorithm, a purely geometric algorithm, and an algorithm which combines ideas from the first two.

Thesis Supervisor: Dr. Robert H. Halstead
Title: Assistant Professor of Electrical Engineering and Computer Science

Table of Contents

Chapter 1. Introduction

1.1 Background

Simply put, the hidden surface removal problem involves creating a realistic computer image of a complex three-dimensional scene. In real life, the hidden surface problem is solved by the laws of physics; objects in front of others are opaque, preventing the light reflected from further objects from reaching the eye. Much research has been devoted to the hidden surface problem, with favorable results; however, each hidden surface algorithm has been designed with a particular class of scenes in mind. Put another way, it has proved very difficult to arrive at a method for removing hidden lines/surfaces which works well for all applications. This has resulted in a large selection of algorithms, each of which is optimized for its own class of scene type and complexity. Each algorithm compromises in some form or fashion; some are designed to produce marginally-realistic images in near real time, while others forsake speed in the interest of producing extremely realistic images.

An overview of research into the hidden-line/hidden-surface problem may be found in [20], and a bibliography on the subject is compiled in [6]. The hidden-line

problem was the first to be researched, due mainly to the prevalence of line-drawing displays. Roberts developed one of the first algorithms [18]. The Warnock algorithm resulted from efforts at the University of Utah to find a real-time algorithm [22]. Many algorithms have been based on calculating the intersections of polygonal edges and faces, notably those of Appel [1], Galimberti [5], Loutrel [13], and Matshushita [14].

Research into hidden-surface algorithms grew in proportion to interest in raster-scan displays and shaded pictures. Romney [19] developed one of the first scan-line algorithms. Watkins [23] developed Romney's work into a practical algorithm and implemented it in hardware. At about the same time, Bouknight developed a scan-line algorithm [2]. Newell, Newell, and Sancha developed a method of priority sorting [15]; this work was further developed by Weiler and Atherton [24]. Fuchs [4] devised a method of using object space decomposition to form priority-trees.

A common vein of sorting runs through all hidden-line/hidden-surface algorithms. Sorting is used to facilitate depth-ordering for objects in a scene, as well as provide other processing orders for elements within these objects (points, edges). Scan line algorithms (such as the Watkins algorithm) generally use a Y-axis bucket sort, followed by an X-axis bubble sort; this facilitates the production of the image data in the same order that it will be displayed on the TV screen. Decomposition algorithms

(notably, the Warnock algorithm) uses a Z-axis sort for all polygonal vertices to provide a priority ordering based on screen depth. For any of the algorithms, the sorting phase consumes a major portion of the processing time. It is for this reason that object space decomposition is investigated, for it offers the potential for removing much of the processing time normally devoted to sorting (hopefully without adding it somewhere else). Object space decomposition is not presented as a panacea for hidden surface problems, since it has limitations which restrict its applicability. However, the approach does occupy its niche in the hidden surface arena, and so is investigated in this thesis.

1.2 Goals of the Thesis

The goals of this thesis are twofold. First, object space decomposition will be explored in detail to evaluate its potential for rapidly determining the priority ordering for objects in a scene. The mechanics of decomposition will be explained and evaluated. A particular approach will be adopted and discussed in detail, including the benefit of the approach, any shortcomings, and improvements to the approach which alleviate the shortcomings. What emerges will be a scheme for using *a priori* information about the scene to provide a quick means for solving the priority ordering problem. The use of decomposition, however, breaks the problem of hidden surface/line removal into two distinct phases, since it totally separates the

determination of the priority of shapes from the calculations which render the scene, based on that priority ordering. Many algorithms combine the two facets.

The second goal of the thesis is to explore algorithms for calculating hidden surfaces and lines using parallel processing. A number of algorithms are evaluated, and while none of them is revolutionary in its own right, the goal is to discover the benefits to be gained by utilizing any concurrency inherent in the algorithms. Some of the algorithms are very simplistic in nature, yet offer substantial opportunity for parallelism. More sophisticated algorithms, however, may reduce the total amount of work required, but might offer less parallelism due to the increased data dependencies. Each algorithm is analyzed in a parallel processing environment, in an effort to evaluate the tradeoffs associated with the algorithm. These analyses suggest how much parallelism can be expected for a given algorithm, and also indicate to what degree the parallelism actually helps in drawing the scene.

The thesis is organized according to these goals. Chapter 2 deals exclusively with decomposition and the establishment of a priority order for objects in a scene. Chapter 3 examines three algorithms for hidden surface removal with respect to computational complexity growth and concurrency issues. All three algorithms assume an existing priority order; as a result, Chapters 2 and 3 are closely coupled. Chapter 4 contains a

summary and recommendations for future work.

Chapter 2. Decomposition

A scene is a collection of objects which we wish to view from an arbitrary position. Object space decomposition is a method of organizing information about the scene which goes beyond simply describing the objects; spatial relationships between the objects are recorded. The basic method involves enclosing the scene in a volume large enough to contain all objects, and subdividing the enclosing space until an easily-interpretable representation of the scene results. A tree is used to map the decomposition, and grows as the space is subdivided. Each node of the tree corresponds to a unique sub-volume. The meaning of "easily-interpretable" is relative in nature, but has to do with minimizing the processing time required to correctly draw the objects in a scene from an arbitrary viewpoint. Rapidly drawing a scene depends on the speed with which the proper ordering of the objects can be determined.

This ordering of objects, or *prioritization*, is usually done using explicit sorting. Using object space decomposition, the priority order may be determined by a simple traversal of the tree which maps the decomposed object space. The traversal order of the tree depends only on the viewpoint, not on the objects in the scene. Thus, for a

changing viewpoint, one could imagine the eye position rotating around the object space, rather than the objects moving within the object space. A basic assumption is that animation of objects within their space is more expensive to process than scenes which are static, since animation requires a re-organization of the decomposition tree each time an object moves. As with most graphics algorithms, the algorithms described in this thesis assume a particular class of applications. The class assumed by this thesis includes any applications in which it is desirable to view a static scene (city, building, satellite) from an arbitrary viewpoint, possibly with near real-time rotation.

Basically, there are two facets to the decomposition process: the method of subdividing the object space, and the halting condition (knowing when the scene is properly decomposed). The subdivision method determines the form of the resulting structure (generally, what kind of tree it is), while the halting condition determines the complexity and size of the structure. This chapter first explores three general methods of decomposing object space: binary subdivision, unequal subdivision, and arbitrary subdivision. Binary subdivision is chosen as the decomposition mechanism, and Section 2.2 explores the method in considerable detail. Section 2.3 discusses the benefits of decomposition with respect to deriving a priority order. Next, two approaches to decomposition, "complete" and "centroid", are discussed. The centroid approach is chosen as the more interesting. The final section explores and resolves a

problem with the centroid approach to decomposition.

2.1 Methods of Subdividing the Object Space

There are three basic methods of subdividing the volume which encloses a scene: binary (equal) subdivision, unequal subdivision, and arbitrary subdivision. The simplest method of subdivision is binary decomposition, whereby each subdivision of an enclosing volume results in eight equally-sized sub-volumes. This means that the cutting planes which define the subdivision are perpendicular to the X, Y, and Z axes, and each plane bisects the volume. The description of the decomposed scene may be stored easily in the form of an octal-tree, where each node of the tree corresponds to a specific sub-volume of the enclosing space. No special knowledge about the scene is assumed; the resultant decomposed scene representation depends solely on the dimensions and orientation of the largest enclosing volume. As will be seen later, the regularity of binary decomposition greatly simplifies the interpretation of the decomposed scene. Figure 2-1a illustrates binary decomposition.

The second method of subdivision is a generalization of the first and involves creating an arbitrary number of sub-volumes from each subdivided volume. A volume may be divided many times along each axis, though each cutting plane is still perpendicular to one of the X, Y, or Z axes. This information may also be stored in tree

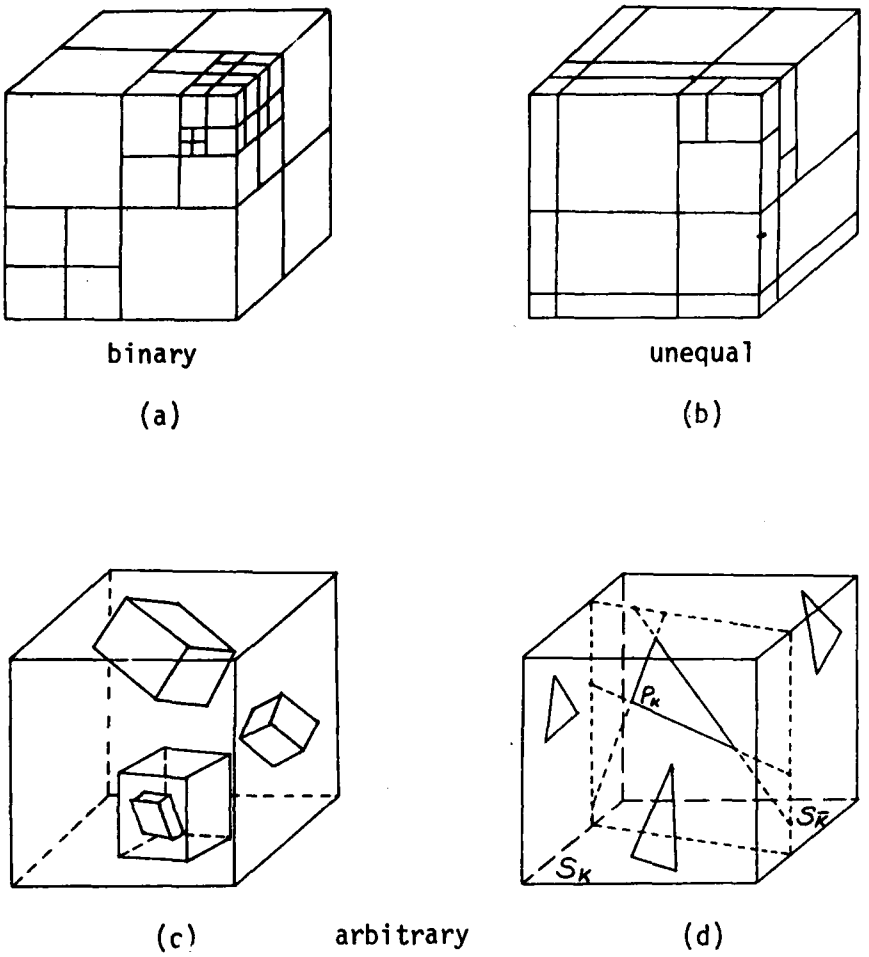


Figure 2-1

Methods of Object Space Decomposition

form, but a great deal more information must be maintained with each node to identify each sub-volume's position and dimensions. With binary decomposition, on the other hand, the position of the node in the octal-tree provides all the positioning and dimension information necessary. This second method of decomposition is sometimes better suited to "neatly" separating scene components from each other, because much more flexibility is allowed in deciding exactly where to divide. Figure 2-1b illustrates this second approach to subdivision.

The third subdivision method generalizes even more than the second method; the volume is not divided along the three major axes, but along arbitrary axes. Reddy [17] developed a method which defines a number of enclosing polyhedra around the shapes in the scene, and provides a maximum degree of freedom in specifying how the scene should be decomposed. Figure 2-1c shows this approach. Fuchs [4] developed an interesting approach to object space decomposition about arbitrary axes. Given a 3-space S and a set of polygons $\{P_1, P_2, \ldots, P_n\}$, polygon P_k is chosen. The volume is divided into two half-spaces, S_k and $S_{k'}$, by the plane in which P_k lies. The remaining polygons are divided between the two half-spaces (clipping polygons where necessary). The half-spaces are distinguished by the use of a positive and negative reference direction for each cutting plane; such a reference is necessary to enable division of the remaining polygons among the half-spaces. The decomposition process continues until

each half-space contains at most one polygon. The resultant data structure is a binary tree, where the left edge of each sub-tree denotes a positive-oriented half-space and the right the negative half-space. The root of each sub-tree contains the polygon whose plane divides the two half-spaces. Because of the problem with the potentially large number of additional polygons resulting from splitting the polygons at half-space boundaries, the algorithm attempts to solve the problem by selecting each polygon P_k such that a minimum number of new polygons will be generated. An inorder traversal of the tree (process one sub-tree, visit the root, process the other sub-tree) generates a priority order for the polygons; at each level of the tree, the subtree processed first is the one which faces toward the eye position (the positive and negative orientations assist in this determination). The importance of this method lies in the characteristic that, for static scenes, the decomposition need only be done once. Changes in the eye position (at image generation time) require only a different traversal order for (some portions of) the tree.

Binary decomposition is rejected in many circles as being wasteful in processor time, too inflexible, and not intelligent enough for many applications. The tradeoffs involved in selecting one method over another are important, and this thesis takes the position that the simplicity of binary decomposition is a very desirable characteristic, since it leads to a very simple interpretation of the scene. Also, processing

considerations are disregarded in this case because the decomposition overhead is assumed to be a one-time cost incurred during a non-critical portion of the applications' lifetime. The scene will be decomposed as it is interactively defined by the user, and the associated structure will be placed on a storage medium until it must be interpreted for use in rendering the scene. At that time, the simplicity of the octal-tree structure will aid in the rapid determination of the priority ordering. For these reasons, this thesis investigates binary decomposition.

2.2 Accessing Information with Binary Decompositon

The simplicity of the binary decomposition method permits a straightforward method for accessing information about the objects in a scene. As mentioned earlier, the decomposed object space may be easily represented in the form of an octal-tree, where each node of a tree is either a leaf or another octal-tree (a parent to eight sub-nodes). A leaf may be "empty" or "occupied." The nodes of the tree at each level correspond to unique sub-volumes. Given a three-dimensional point (x,y,z) in object space, the node (volume) containing that point can be determined simply from the coordinates of the point expressed as 10-bit numbers, assuming a range of 0-1023 in X, Y, and Z [17]. Concatenating the high-order bit of each coordinate into a three-bit number (X most significant) yields the node at the first level of subdivision. Within

that sub-volume, concatenating the next-highest bit in each coordinate identifies the node containing the point at the second level of subdivision, and so on, until the concatenated low-order bits of each coordinate defines the node of highest resolution (one pixel on each side). This method is valid only for cubic volumes with dimensions equal to some power of two.

For cases in which the enclosing volumes are not cubic or the dimensions are not a power of two, special handling is required. One method is recursive in nature and involves testing the input point (x,y,z) against each of three cutting planes to determine which of eight octants the point lies in. [In this thesis, *octant* and *quadrant* refer to three- and two-dimensional regions, respectively.] The cutting planes are defined by bisecting the volume along each axis into two equal parts. The method recurses to process the current octant by defining three more cutting planes, and continues to recurse to an arbitrary depth. The procedure below illustrates the subdivision process. It accepts as input a test point (x, y, z), the origin (x0, y0, z0) and size (sizex, sizey, sizez) of the volume to be subdivided, and the current depth of decomposition. At each level of decomposition, the procedure identifies the quadrant that contains the test point, according to the convention in Figure 2-2. Notice that this method works equally well with cubic volumes whose dimensions are a power of two.

```
function Octant (x, y, z, x0, y0, z0, sizex, sizey, sizez, depth)
    cutx = x0 + sizex/2
    cuty = y0 + sizey/2
    cutz = z0 + sizez/2

    Q = 0
    if x >= cutx then
        x0 = cutx
        Q = Q + 4
    endif
    if y >= cuty then
        y0 = cuty
        Q = Q + 2
    endif
    if z >= cutz then
        z0 = cutz
        Q = Q + 1
    endif
    print Q      /* or save it in some form or fashion */
    if depth < max then
        Octant (x, y, z, x0, y0, z0, sizex/2, sizey/2, sizez/2, depth + 1)
    endif
end Octant
```

As another example of the interpretation provided by an octal-tree representation, consider the case in which an object is added to a scene represented by an octal-tree. It might be desirable to know whether a given point in the new object is contained inside an octant which is empty or is already occupied by another object; this condition of two objects occupying the same space is reason for further decomposition of that space. To make that determination, traversal of the octal-tree is necessary, and continues until a

leaf node is reached. At that time, it is possible to tell whether the octant corresponding to that node is empty or occupied.

As an illustration, assume that we have an object [a single point (x,y,z)] which is to be added to an existing object space represented by an octal-tree. We are interested in whether a collision will occur (whether the point would share a node of the tree with another point). If so, then that node would need to be decomposed further, since we assume that each point must occupy a unique sub-volume. Thus, given the coordinates of the new point, we would like to be able to find a leaf node which is either empty or already occupied by a point. This can be done by expressing the coordinates of the point as binary numbers, as described earlier. As an example we will use the point (15,453,890) and assume an enclosing volume with origin (0,0,0) and dimensions of 1024. The coordinates of the test point are expressed as ten-bit numbers as follows.

decomposition level
(tree depth)
---------->

X 0000001111 15
Y 0111000101 453
Z 1101111010 890
Q 1323115656

The decomposition can be seen all the way down to the "unit volume" level by grouping the corresponding bits for each coordinate into ten three-bit numbers, each

corresponding to the volume as pertains to the convention in Figure 2-2. For this example, the decomposition would proceed in the order of volume 1, then volume 3 of 1, then 2 of 3 of 1, and so on for sub-volumes 3, 1, 1, 5, 6, 5, and 6. This establishes a road-map which can be used to search down through the octal-tree, to the maximum depth if necessary, until a leaf node is found. This node may be empty, in which case the point could occupy the corresponding volume. However, if a point already existed in the volume, further decomposition would be required until such time as each point had its own volume. Using this methodology of decomposition down to the single unit resolution, a problem naturally arises when two identical points exist. In such a case, an infinite number of subdivisions would theoretically be necessary before each point could occupy a unique volume. There are basically two methods of handling such a situation: the first would involve signalling an error condition if two points were identical (the situation would not be allowed), while the second would involve the detection of identical points followed by the inclusion of appropriate information in the data base flagging the special condition. The first alternative seems to be unduly restrictive on the flexibility of a modelling program, while the second involves tradeoffs both in storage space and processing time (when a view of the scene is interpreted). It will be shown later that a proper choice of the halting condition for decomposition can make such a situation unlikely.

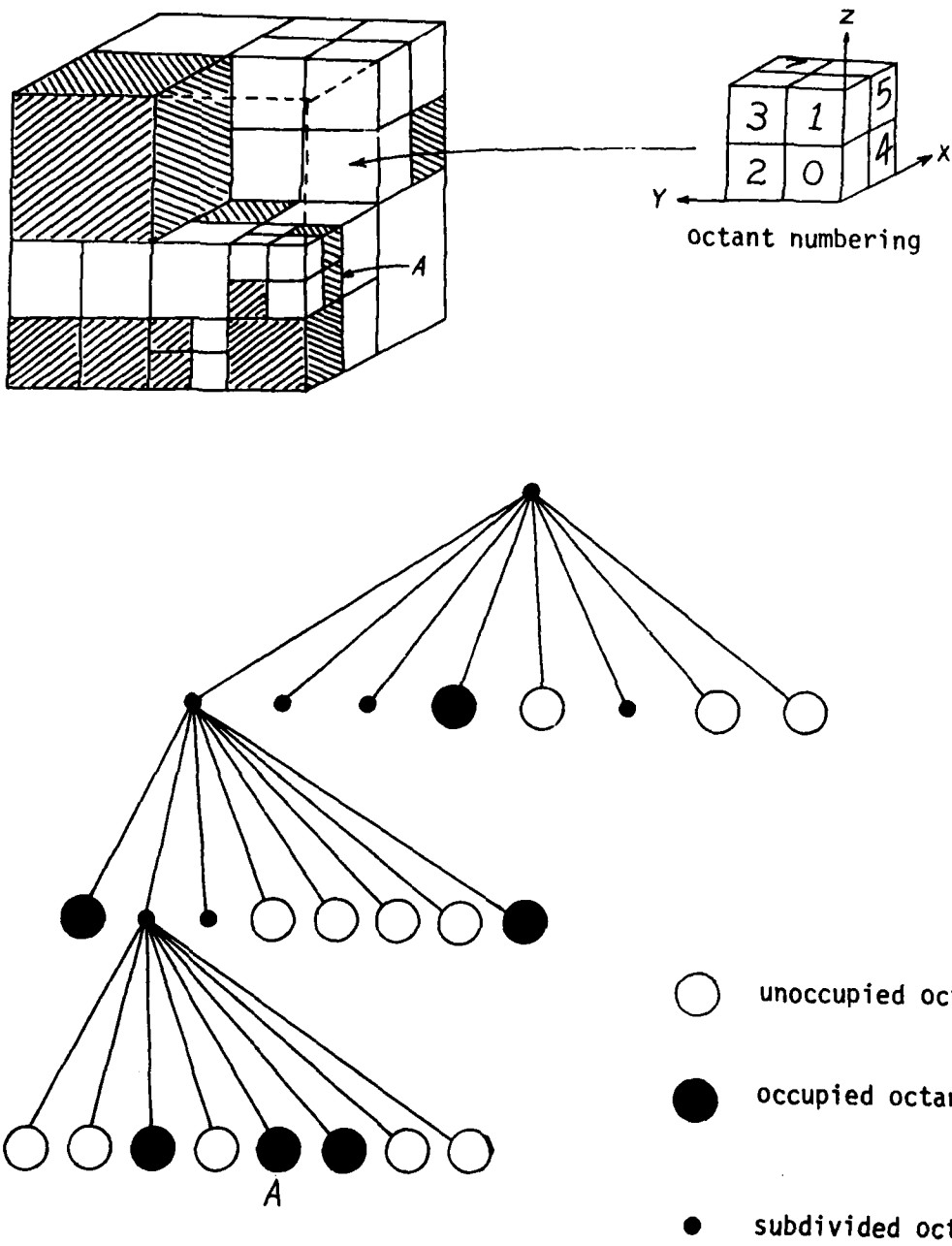


Figure 2-2

Octal-trees

Given a node in the octal-tree and the coordinates of the enclosing volume's origin, the boundaries of the volume corresponding to that node may be determined very simply [11]. The dimensions of the sub-volume (D) are each $S*2^{-n}$, where S is the size of the largest enclosing volume, and *n* is the level in the tree at which the node resides. The origin of the sub-volume can be determined from the position of the node within the tree. For instance, in Figure 2-2 the node marked A is mapped 0-1-4 in the octal-tree (that is, node 4 of node 1 of node 0 of the tree). This corresponds to the numbering convention for the volumes as shown in the upper right corner of the figure. Assuming dimensions of 1024 units, the coordinates of the origin of the sub-volume may be calculated by the formula

$$X_{min} = \sum_{i=1}^{n} X_i \, (size/2^n)$$

Y_{min} and Z_{min} are calculated accordingly. X_i means "the ith bit of the X part of the map" (most significant bit of the node position at each level). For the above example, the node map may be viewed as

Q --	X	Y	Z
0 --	0	0	0
1 --	0	0	1
4 --	1	0	0

Then the origin of the sub-volume (X_{min}, Y_{min}, Z_{min}) may be calculated as

$$X_{min} = 0*512 + 0*256 + 1*128 = 128$$

$$Y_{min} = 0*512 + 0*256 + 0*128 = 0$$

$$Z_{min} = 0*512 + 1*256 + 0*128 = 256$$

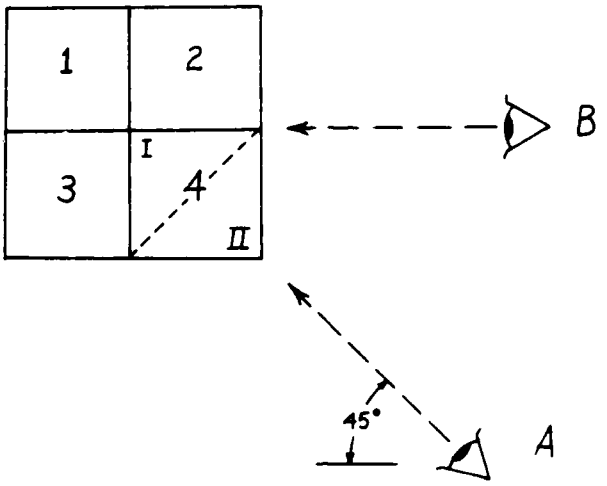
Thus, the origin of the node shown is (128, 0, 256), with size of $1024/2^3$, or 128.

2.3 Tree Traversal versus Explicit Sorting

The hidden surface removal problem typically involves a great deal of sorting. Depending on the algorithm, all points in the scene are sorted in one or more of X, Y, or Z order. This sorting consumes a great deal of computation time, and it has been suggested that future research into the hidden surface problem concentrate on better methods of sorting [20]. Binary decomposition of a scene would result in a decomposed scene which would require no explicit sorting of the object points. This is due to a characteristic of the decomposed scene; if the eight main octants of the volume (first level of the octal-tree) are sorted in order of nearness to the viewer, this ordering of the octants defines the traversal order of the octal-tree at all levels. This statement points out an essential characteristic of binary decomposition, and should be discussed in detail to demonstrate its validity.

Consider the situation shown below, with four quadrants in two dimensions. The discussion for this simplified scenario applies equally well for the three dimensional

case of eight octants.



From position A, quadrant 4 has top priority, quadrants 2 and 3 are next (with equal priority), and quadrant 1 has the lowest priority. This ordering is based upon the distance from the eye position to the center of each quadrant. Situations in which quadrants have equal priority happen only when the viewing angle is a multiple of 45 degrees; in these cases there is no overlap between those quadrants, and an arbitrary selection of one quadrant over another will produce a correct view. Thus, from position B, the quadrant-pair 2/4 takes priority over pair 1/3, though within each pair it does not matter which quadrant is treated first. Going back to position A, though, it is important

to understand the relationship of any objects in quadrant 4 to those in quadrants 2 or 3, since that relationship forms the basis for establishing the priority order of objects based solely on traversal of the octal-tree. Assuming for the moment that objects are single points, it is the case that every object in quadrant 4 is either in region I or II, separated as shown by the diagonal line). Every object in region II is closer to the eye than any object in quadrants 2 or 3. Note that the line separating the two regions will not always intersect the corners of the quadrant; it depends on the viewing angle. It is possible, however, that objects within region I are farther from the eye position than objects in the other two quadrants. Consider Figure 2-3 which shows object A in the back corner of quadrant 4 and object B in the front corner of quadrant 3. Object A is farther from the eye position than B, and would normally have a lower priority, except for its being in a quadrant which has higher priority than B's quadrant.

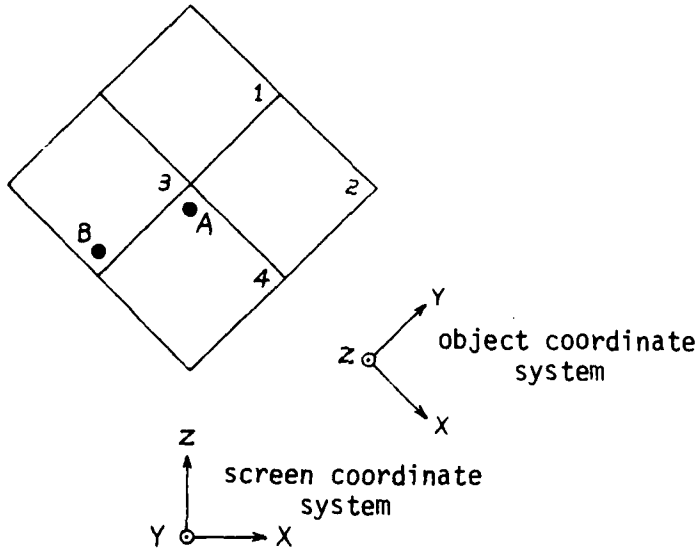


Figure 2-3

Quadrant Sorting versus Explicit Sorting

Assuming a separation in object space between A and B of DX and DY, the distance perpendicular to the viewing axis between A and B in image space becomes

$$DY\sin(\text{theta}) + DX\cos(\text{theta})$$

where *theta* is the angle through which the eye position is rotated (in this case, in object space about the Z-axis). This distance is of course a minimum when the two points coincide in real space, and it increases as a function of the points' separation. Thus, even though object B in quadrant 3 is closer to the eye than object A, the latter may be drawn first, without sacrificing accuracy, because of the resultant separation. This

relationship is true of any objects within quadrant 4. Some of the objects may have higher priority than those in either of quadrants 2 or 3, but any objects which are farther away are guaranteed to be separated in image space such that no incorrect drawing could possibly occur. This reasoning points to the requirement for physical decomposition. In order for the discussion above to hold true, an object must be completely contained within a quadrant. With the assumption that points are objects, that requirement is trivially satisfied. For polygonal objects, however, it is possible that the polygons may cross quadrant boundaries. In such cases, the polygons must be physically divided at the boundaries. The case for physical decomposition will be examined shortly in more detail.

Of course, if there are several objects in quadrant 4, then some means of prioritizing those objects must exist; this means that the quadrant needs to be sub-divided at least one level further. The depth order relationship of these sub-quadrants, with respect to the viewing axis, will be identical to that of the four main quadrants; that is, within quadrant 4, sub-quadrants 1 through 4 will be ordered 4,2/3,1. This is another feature of binary decomposition; the symmetry of the quadrants allows a generalization of the priority order based on the four largest quadrants. As a result, any object within sub-quadrant 4 of main-quadrant 4 takes priority over any objects in 2 of 4, 3 of 4, or 1 of 4, all of which take priority over the

objects in the other three main quadrants. This establishes a traversal order for the structure defining the scene; each node is processed (in order according to the main level priority) by completely traversing its sub-tree, and within each sub-trec the same strategy holds true. This "flattening" of the tree through traversal will yield a prioritized list of the objects in the scene. The algorithm for the traversal is defined below.

```
/* Assume:  a list 'sort-order' which identifies the octants in priority order */

procedure Process (node)
 for each octant in sort-order
        if node[octant] is subdivided then
            Process (node[octant])
        else
        if node[octant] is an object then
            prioritized-list = prioritized-list + node[octant] /* + = catenation */
        endif
    endfor
end Process
```

With the scene decomposed such that each volume contains a very simple amount of information (only one object), each object may be drawn as it is encountered in the tree traversal process. This is essentially the same characteristic as the binary partitioning tree of [4]. For a static scene, then, different viewpoints of that scene may be generated simply by applying the necessary transformations to the main enclosing volume and

sorting the eight major octants in order of nearness to the viewer. Notice that just as a list of values can be sorted in either ascending or descending order, so can the octal-tree traversal yield a list of objects prioritized in order of increasing nearness to the viewer or distance from the viewer.

2.4 Complete Decomposition

In this paper, "complete" decomposition requires that only one object occupy any octant in the subdivided object space. For purposes of discussion, we assume these objects to be arbitrary convex polygons. This method of decomposition adheres to the simplifying assumptions of the previous section, and results in a simple interpretation of the polygons in the scene. As a result, the establishment of a priority ordering for those polygons becomes simply a tree traversal. A decomposition philosophy of allowing only one polygon per octant dictates that the polygons themselves may have to be physically subdivided at the octant boundaries. This is necessary because the position of a node within an octal-tree implies a priority ordering only for the polygon within the node's boundaries, not for any portion of the polygon which might extend beyond the boundaries. As a result, polygons must be "clipped" to fit completely within a sub-volume. This physical decomposition of polygons illustrates one of the tradeoffs between equal (binary) and unequal subdivisions. With unequal subdivisions, it might

be possible to subdivide the octant such that no subdivision of polygons is required. This is because the unequal subdivision algorithm has a great deal of flexibility with regards to where the cutting planes are placed (and in fact, if they are used at all for any given axis). However, such flexibility is not characteristic of the binary decomposition technique. It is likely that a polygon may intersect one or more of the cutting planes which subdivide an octant. Each intersection of a convex polygon with a cutting plane results in two new points and two new polygons (the old polygon in effect is discarded).

The process necessary to subdivide a polygon is in effect a three-dimensional clipping algorithm. The polygon need be tested only against the cutting planes which subdivide the volume, and not against the boundaries of the volume itself; this is because any polygons inside the current volume are a result of applying the clipping algorithm for the larger volumes which contain or border on the current volume. Determination of an intersection of a polygon with a cutting plane is simple; the only information required is the location of the cutting plane (e.g., the plane defined by the equation $X = 10$), and the three-dimensional coordinates of each vertex in the polygon, which in effect define the lines (edges) of the polygon. For each edge of the polygon, an intersection exists if the endpoints of the edge lie on opposite sides of the cutting planes. If so, the coordinates of the intersection point are calculated based upon the DX, DY, and DZ of the edge. Each edge is tested in this manner, until two intersection

points are found. Using these new points, two new polygons are defined, and each polygon is then tested against any remaining cutting planes. For the simplest case (convex polygons), the worst case for a subdivision is nine new points created and seven new polygons (see Figure 2-4), all of which must be maintained in the data structure describing the scene. These new polygons are hereafter referred to as "artificial" polygons.

As the polygons become smaller through successive division, truncation or roundoff errors could result in anomalies in the subsequent depiction of the surface represented by the large polygon, due to differences in the plane equations for the smaller sub-polygons. This would tend to manifest itself in the form of surface normals with slightly different directions, and could result in shading anomalies.

To gather statistics on this problem, a test program was developed to take an arbitrarily-oriented polygon and decompose it through five or more levels (the octal-tree became five or more levels deep). This resulted in a large number of new points and polygons. Each polygon was processed to calculate a unit normal vector for the polygonal surface; the cross product of the first two edges was the convention. Each surface normal was compared against the normal of the original polygon by computing the angle between the two vectors (arccosine of the dot product). It was found that the

error averaged 17 degrees, with a maximum of 36 degrees. The coordinates were represented as "single precision" floating point numbers with 24 bits of fraction.

While the actual error values are not significant in themselves, it points out that a large amount of subdivision results in polygons so small as to produce surface normals of wide variation. A better way of handling this condition would be to save the appropriate information for the original polygon, then have all sub-polygons point to that information. This information would most likely include three vertices of the original polygon (for calculating the surface normal) and any shading or surface detail information. Such sharing of information between polygons would also be advantageous from a data storage point of view. However, the problem of the large number of "artificial" points and polygons remains. Crowded scenes which result in many collisions within octants would consume a great deal of storage space, as well as processing time to access the additional structures.

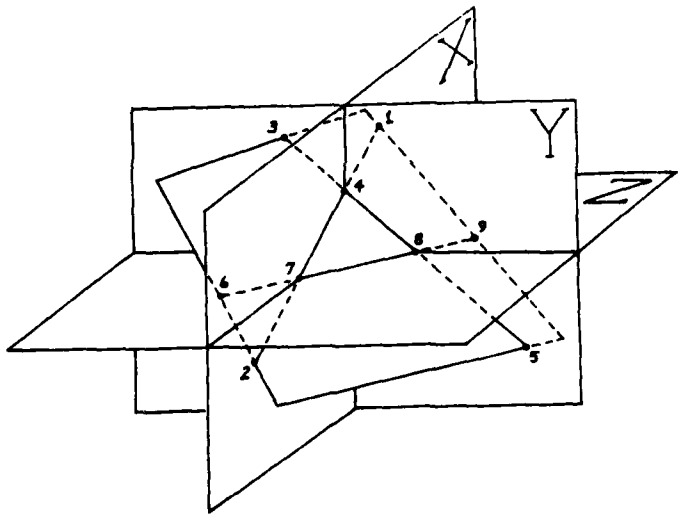


Figure 2-4

Division of a Convex Polygon by X-Y-Z Cutting Planes

Also, for situations in which the scene complexity is high (intersecting polygons or simply shapes with common faces), an infinite number of subdivisions would be necessary to satisfy the halting condition. Handling such complex relationships requires a compromise in which the subdivision halts after a pre-defined number of iterations, and special information is saved which flags the octant as containing more than one polygon. Better still, it would be desirable to determine surface intersection or common faces as soon as possible, to avoid any pointless decomposition. This could save a significant amount of octal-tree storage space, as well as processing time. In cases where octants did contain one or more polygons, special logic would have to be invoked for

each view of the scene in order to prioritize the shapes. All in all, the requirement of one polygon per octant seems to be too restrictive, for even though it implies a simple "traverse and draw" approach to the octal-tree processing, there will exist so many artificial polygons in the scene that the computational savings from the simple interpretation will be engulfed by the processing dictated by the additional objects in the scene.

2.5 Centroid Decomposition

One response to the problems with complete binary decomposition is to greatly simplify the halting condition for the decomposition process; a scene is considered to be properly decomposed when each octant contains not one polygon, but one "centroid" for a shape in the scene. The term "centroid" refers to the center of the polygon, and could be viewed simply as the center of mass for the polygonal vertices. The location of the centroid is actually arbitrary, and need not be consistent among the polygons in a scene. In fact, the method of choosing a centroid only affects efficiency, since careful selection of a centroid for each object could affect the amount of decomposition required Even for the simplest centroid selection method, however, centroid decomposition implies a much smaller amount of subdivision, both with regard to the enclosing volume and with regard to the objects themselves. In fact, no polygons must

be physically subdivided. The decomposition process is guaranteed to halt as well, except in such cases as the user decides to define two or more polygons with the same centroid. (Such situations are disregarded in this analysis).

A result of decomposition based on centroids is that the shapes may extend past octant boundaries. One tradeoff of this centroid method is that the octal-tree is not as easily interpreted as with the previous method (one polygon per octant). In fact, this more complex interpretation is a very important characteristic, because it forms the basis for anomalies in the prioritization when the octal-tree is traversed. Such problems, however, are surmountable. The algorithm for performing the centroid decomposition is presented below as a pidgin Algol procedure which builds the scene one object at a time (thus tailored to an interactive approach). A centroid is allowed to occupy any octant of the defined object space, as long as that octant is not already occupied and is not subdivided. A polygon and its centroid are associated with the same node in the octal-tree.

```
/* In this routine, a "shape" is a data type containing a centroid and */
/* a number of polygons. The centroid is defined as a triple (x, y, z) */

procedure AddShape (shape, x, y, z, size, node)
    centroid = shape.centroid
    xmid = x + size/2
    ymid = y + size/2
```

```
    zmid = z + size/2

/* The octant number can be expressed as a three-bit number, with bit 0 */
/* representing the Z-axis boundary, bit 1 the Y-axis boundary, and bit 2 */
/* the X-axis. Each bit is set (by additions of powers of two) depending */
/* on whether the point is past the midpoint boundary of the volume with */
/* origin (x,y,z) and dimensions 'size'. */

    octant = 0
    if centroid.x > xmid then
        octant = octant + 4
        x = xmid
    endif
    if centroid.y > ymid then
        octant = octant + 2
        y = ymid
    endif
    if centroid.z > zmid then
        octant = octant + 1
        z = zmid
    endif

/* If node is empty, go ahead and add the shape. */

    if node[octant] = 'empty' then
        node[octant] = shape
    else

/* If node is already occupied, divide the octant and re-process both shapes. */

    if shape? (node[octant]) then
        new-node = MakeNode
        AddShape (shape, x, y, z, s/2, new-node)
        AddShape (node[octant], x, y, z, s/2, new-node)
        node[octant] = new-node
    else

/* Node is subdivided; RECURSE */
```

```
        AddShape (shape, x, y, z, s/2, node[octant])
    endif
end AddShape
```

2.5.1 Problems with Centroid Decomposition

Using a halting condition of one centroid per octant, the sorting of the eight major octants of the enclosing volume and the subsequent traversal of the octal-tree in effect performs a priority sort of the shapes based on the centroids. Using the octal-tree to accomplish this prioritization provides a savings in computation time over a simple sort of all the centroids in the scene. Traditional sorting methods require time anywhere from $O(N^2)$ for a bubble sort to O(NlogN) for a merge sort or $O(\log^2 N)$ for Batcher's method [12], where *N* represents the number of objects being sorted. (It should be pointed out that Batcher's method requires O(N) processors.) Traversal of an octal-tree can be done in linear time O(N'), where *N'* is the number of nodes in the tree. The relationship between *N* and *N'* can be important, since for an octal-tree the ratio of empty nodes to nodes containing objects is usually greater than 1:1. In practice, the ratio will depend both on the separation between the centroids of the objects in the scene, and on the positions of the centroids within the enclosing volume. Obviously, the ratio should be as low as possible in order to take advantage of the linear growth characteristics of octal-tree traversal.

One important difference between sorting and octal-tree traversal is that the traversal can be used in a "stream" or "producer-consumer" mode. That is, the traversal occurs in priority order and need not finish before the information from the traversal is used. Even with Batcher sort, the order is not guaranteed to be correct until the final merge takes place. Generally speaking, however, the first non-empty octal-tree node encountered in the traversal is guaranteed to have top priority, and subsequent non-empty nodes can be processed as they are encountered. Thus, the maximum time necessary to produce the first object from the octal-tree is O(q), where q is the maximum depth of the tree.

There is another important way in which the use of an octal-tree is not completely analogous to the situation of sorting the centroids of all shapes in a scene. The halting condition for centroid decomposition merely requires a centroid to be alone in an octant; it does not imply anything about the relative positioning of shapes whose centroids are in different octants. For instance, Figure 2-5 shows the situation (in two dimensional projection) in which shape A (whose centroid lies in quadrant 4) and shape B (in quadrant 1) is viewed from the eye position shown. According to the quadrant sort, quadrant 4 has a higher priority than quadrant 1, yet if the centroids were sorted, shape B would have a higher priority than shape A. Since the shapes are drawn according to the priority ordering of the quadrants, it is conceivable that an erroneous

picture could result, meaning that some shapes behind others might appear to be in front.

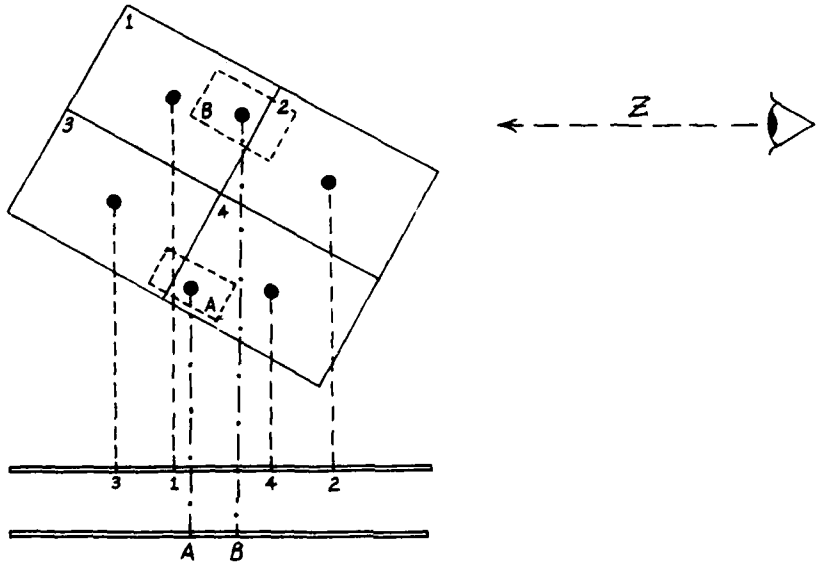


Figure 2-5

Centroid Sorting versus Quadrant Sorting

2.6 Coupling of Shapes

As mentioned above, prioritizing the scene by traversing an octal-tree can conceivably cause the hidden surface algorithm to produce an erroneous picture. This is due to the fact that the centroid's position in the scene (its position in the octal-tree) does not contain any information about the shape's size, orientation, or proximity to

other shapes. Such information is extremely important because in many cases the relative position of two centroids is simply not enough information on which to base a priority decision. Problems in scene depiction are characteristic of the use of centroids as a basis for priority ordering, and have nothing to do with the use of an octal-tree or object space decomposition. Figure 2-6 illustrates a simple situation (in two dimensions) in which the centroid for shape A is closer to the viewer than the centroid for shape B; however, shape B should actually be drawn as being "in front" of shape A. In such cases, more sophisticated depth testing must be performed in order to correctly draw the objects. Such testing can be very time-consuming, and must be minimized in order to hasten the processing of the scene; situations requiring the additional processing should also be the exceptional case rather than the rule. For this reason, we must be able to determine if two shapes are logically "coupled" to each other.

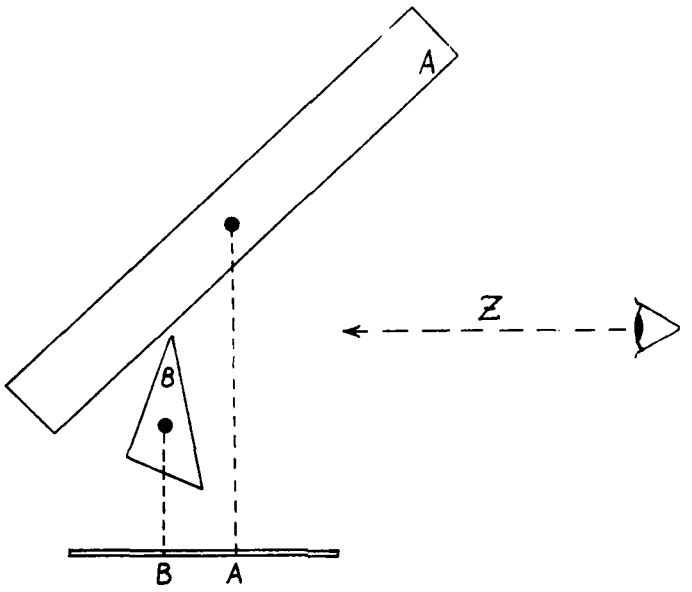


Figure 2-6

Problems with Centroid Sort

Coupling denotes a condition in which the hidden surface algorithm must consider not only the centroid priority ordering for any given view, but must perform an additional test which directly compares a shape against any other shapes to which the shape is coupled. Scene definition would thus involve an algorithm for determining and flagging coupling, and the tree traversal process would "uncouple" shapes as necessary. This uncoupling process could cause shapes to be drawn out of order in relation to the centroid priority sort. However, no additional processing is necessary (other than the test). One shape merely "cuts in line" ahead of the others.

Technically, it would be possible to completely couple a scene so that no sorting would be necessary. Complete coupling establishes tests between each pair of objects; for N objects in a scene, the combinatorial C(N,2) yields the number of tests which must be performed to determine the priority ordering for the scene. This is $O(N^2)$ complexity. It is likely that objects sufficiently removed from one another could be drawn using a simple centroid depth test. So, the choice of an algorithm to determine and resolve coupling could greatly affect the computational time required to resolve ambiguities. Any coupling algorithm would seem to consist of two sub-problems: the determination of situations in which coupling is necessary, and the formulation of a fast and effective test for deciding which shape should appear "in front" of another.

2.6.1 Determination of Coupling

A two-dimensional situation which dictates coupling is illustrated in Figure 2-7a; the situation in three dimensions is analogous. As shown in the figure, coupling exists in the situation in which the centroids are equidistant from the viewer, AND there is overlap between the shapes. Overlap can be defined simply as the situation in which any line drawn from the eye position to a vertex of one shape intersects an edge of the other shape. In such situations, it is possible that if the shapes are rotated slightly in the plane of the paper, one centroid will be closer to the viewer, while in fact that shape

should actually be of lower priority than the other shape (Figure 2-7b). In fact, this is the same situation as with the objects in Figure 2-6. Stated another way, if two shapes have centroids equidistant from the viewpoint and one shape cannot be arbitrarily selected over the other without causing an incorrect rendering of the scene, then coupling exists. Clearly this is the case in Figure 2-7a, since arbitrary selection of shape B over shape A will result in a correct view (drawing in a front-to-back order), but selection of shape A over shape B will not.

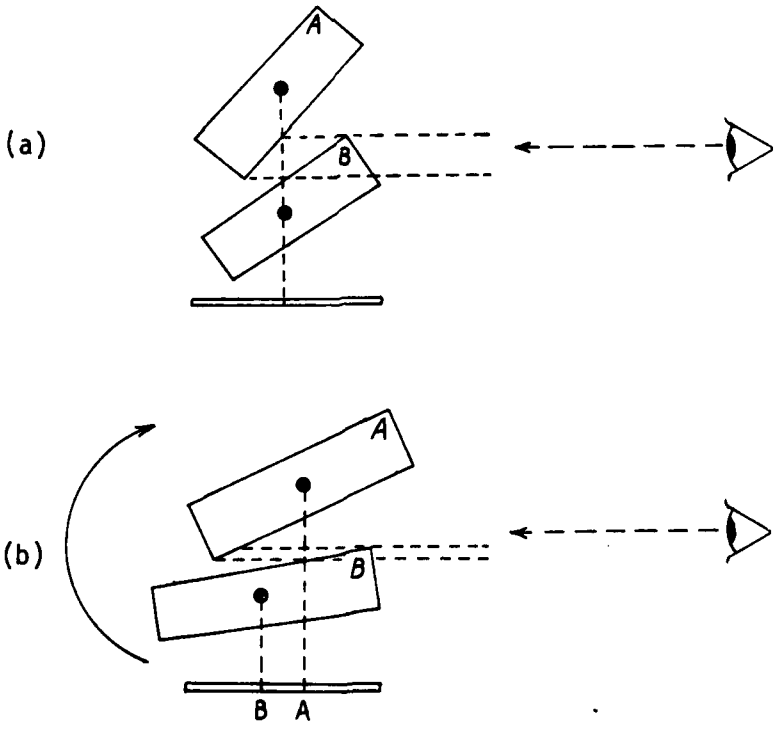


Figure 2-7

Coupling Necessary when Overlap Exists

The tricky part of the coupling problem is that it is completely dependent on the viewpoint selected. If coupling exists for any viewpoint, it should be flagged as such and an effective test must be devised for resolving the confusion. In three dimensions, the problem is complex because it may be the case that for an arbitrary two-dimensional projection of two shapes, coupling appears to be in order. Yet, as seen in Figure 2-8, another projection shows that coupling is actually not necessary because the shapes are separated in space enough so that the centroid priority ordering is sufficient.

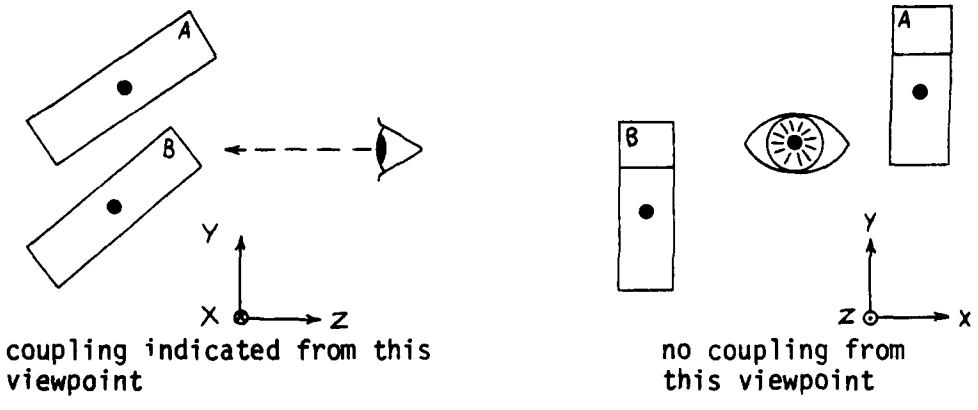


Figure 2-8

Coupling Depends on Viewpoint

Since it is desirable to minimize coupling, it would be helpful if the algorithm

could eliminate any unnecessary coupling. An algorithm for determining coupling in three dimensions would be: *project each polygon onto a segment along the line that connects the two centroids. If the two segments overlap, than coupling exists between the polygons.* An implementation of this algorithm will now be presented for discussion purposes later in this section.

For the two shapes in question, define an axis which connects the two centroids. Translate the axis and rotate it so that the first centroid is at the origin and the axis connecting the centroids is parallel with the X axis. The algorithm for determining the transformation matrix is presented below, and is based on Figure 2-9 below.

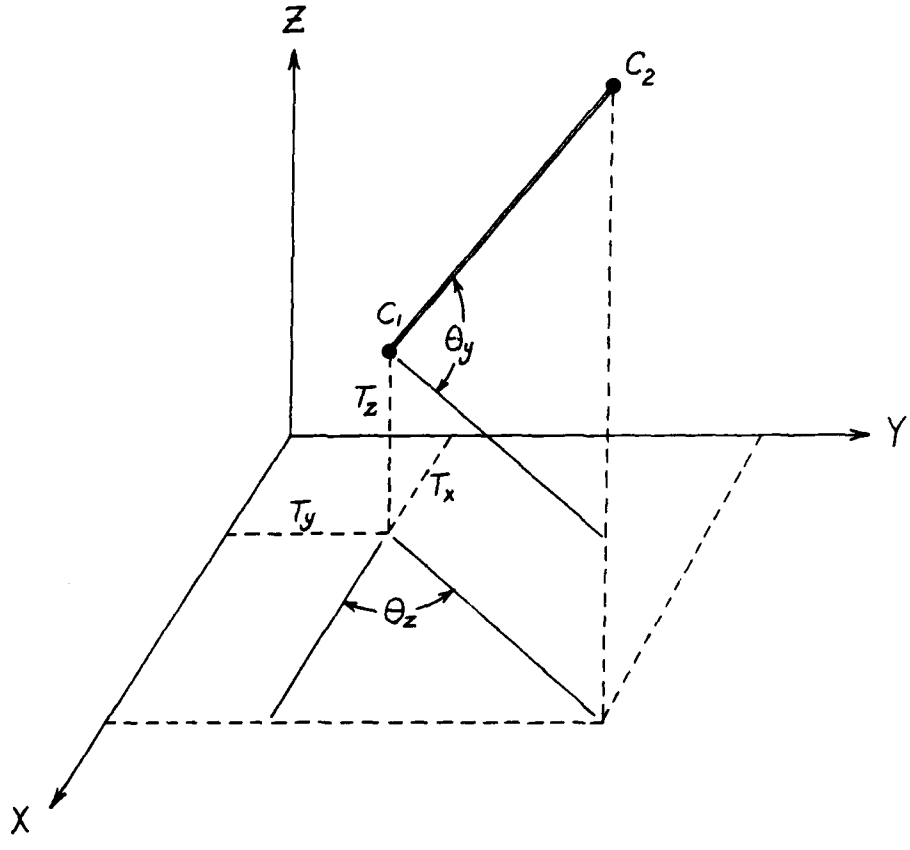


Figure 2-9

Rotations and Translations for Centroid Axis Alignment

```
function Get_Matrix (p1 p2) returns Matrix
 dx = p2.x - p1.x
 dy = p2.y - p1.y
 dz = p2.z - p1.z

 d1 = sqrt (dx*dx + dy*dy)
 sin1 = dy / d1
 cos1 = dx / d1
 d2 = sqrt (dz*dz + d1*d1)
 sin2 = -dz/d2
 cos2 = d1/d2

 matrix1 = Translate (-p1.x, -p1.y, -p1.z)
 matrix2 = Rotate_Z (sin1, cos1)
 matrix3 = Rotate_Y (sin2, cos2)

/* The magical operator * denotes matrix multiplication */

 return (matrix1 * matrix2 * matrix3)
end Get_Matrix
```

Once the proper matrix is derived, transform all points for the shapes. That being done, put the X-components of each point through a "min-max" test. If it is the case that overlap exists between the shapes (the min or max extent of one shape falls between the min and max of the other), then coupling is necessary. Figure 2-10a illustrates a case in which coupling is dictated, while Figures 2-10b and 2-10c illustrate situations which show no need for coupling.

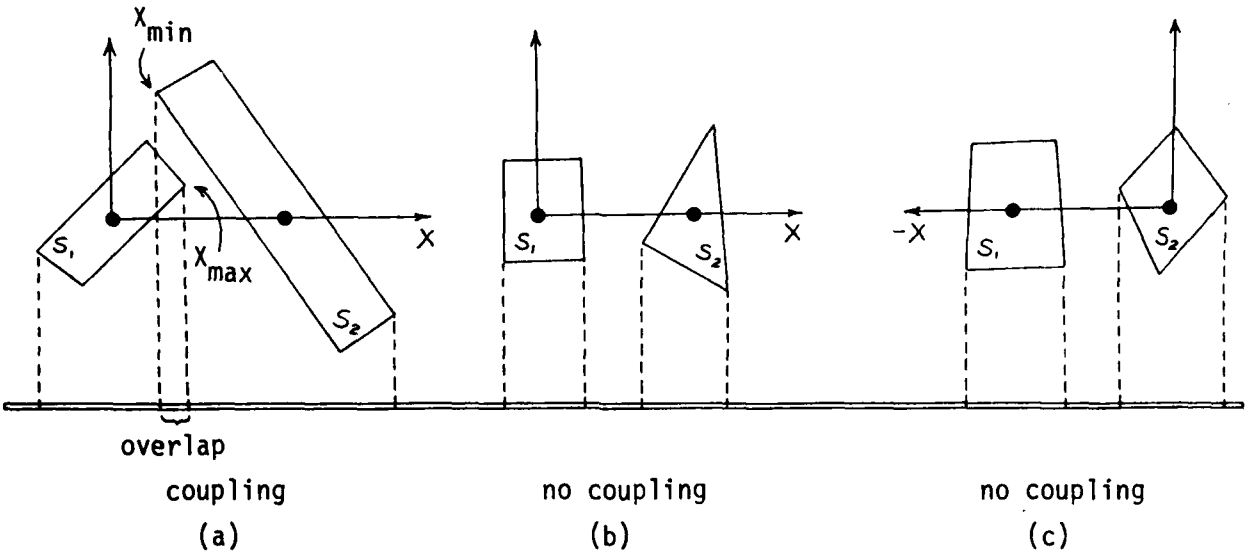


Figure 2-10

Three Coupling Situations

For any given shape, all other shapes need not be tested for coupling; it is likely that shapes may be quickly eliminated from consideration by first using a min-max test to determine if two shapes are sufficiently removed from one another. This involves bounding each shape in a box defined by its minimum and maximum values in X, Y, and Z. If the two boxes do not intersect, then no test for coupling need be performed. Such a test is extremely fast, and shapes may be eliminated in parallel. In fact, it would probably be useful to perform a secondary level of decomposition while defining a scene, one that keeps track of which shapes cross into which octants. Only shapes which extend into the same octant need be considered for coupling to each other. Once

the scene is completely defined, this secondary information would be discarded.

2.6.2 Growth in Complexity

One area of considerable concern is the complexity growth of the coupling in a scene as the number of objects increases. As mentioned earlier, it would be possible to couple all shapes in a scene, resulting in an $O(N^2)$ growth in the processing required to decouple shapes. In scenes with a very large number of objects (a representation of New York City, for instance), one would hope that the amount of coupling grows in a near-linear fashion with respect to the number of objects. In fact, this is the case, assuming no penetration of shapes. The definition of coupling implies a proximity condition which must be satisified before coupling is possible. Taking the min-max test one step further, then for two shapes to be coupled, there must be intersection of two enclosing spheres (where the center of each sphere is the shape's centroid, and the radius is the maximum distance to any of the shape's extremities). This suggests that the optimal choice of centroid is that which minimizes the radius of this sphere.

For scenes containing objects of somewhat near the same volume and no penetrating shapes, there are a limited number of shapes which can penetrate a given shape's enclosing sphere. Thus, as the number of objects in a scene increases, so does the likelihood that a given object will lie outside the enclosing sphere of another object.

As a result, it is unreasonable to consider a scene in which every object in the scene is close enough to every other object to require coupling, at least for a large number of objects. It is only in such a case that N^2 growth would appear. The coupling complexity is more likely to resemble some sort of arbitrary three-dimensional network or matrix in which each shape is potentially coupled to all shapes immediately around it, but not to any shapes "outside its reach."

2.7 Decoupling of Shapes

The second phase of the coupling problem involves the determination of a fast and effective test for resolving the conflict between two coupled shapes. Several courses of action are available, but the underlying concern is one of speed. Any test must involve a minimum of computation so as to determine as rapidly as possible each shape's relative priority. The simplest method might be to find a point on one of the shapes which would be tested against a point on the other shape; the relative positioning of those points would determine the priority for the two shapes. Another method might involve the specification of a surface normal for one of the shapes' faces. This surface normal would be calculated for each view, and its direction (in the eye coordinate system) would determine the priority ordering for two shapes. For instance, in Figure 2-11, the polygon labelled 'A' is selected as the reference polygon. When the

view for the scene is constructed, the surface normal for the polygon is calculated. If the normal vector has a component (usually Z, by convention of the eye coordinate system) which points toward the eye position, then the shape containing polygon A should appear to be behind the other shape. Otherwise, the converse is true.

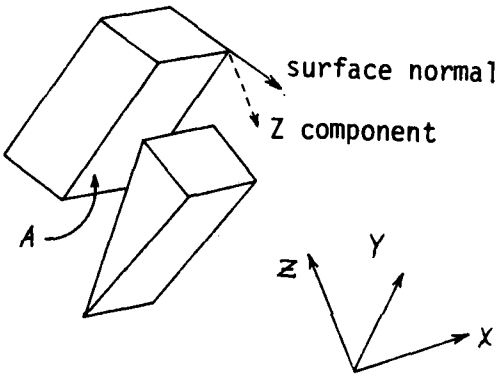


Figure 2-11

Establishment of Reference Polygon to Resolve Coupling

Another method found to be particularly fast and effective involves the establishment of a reference vector which "looks between" the shapes in question. Consider two shapes, S_1 and S_2. When testing for coupling as discussed in section 2.6.1, the first shape (S_1) is always transformed in the coupling-test coordinate system such that its centroid lies at the origin. S_2 is transformed and rotated so that its centroid lies on the X-axis of that same coordinate system. If coupling exists, it is because of the

overlap illustrated in Figure 2-10. If a line is defined by connecting the X_{max} point of S_1 with the X_{min} point of S_2, this defines a viewing line which effectively looks "between" the two shapes. Viewing the two shapes along this line results in an overlap of only two points. Changing the viewing angle slightly would mean that one of the two shapes was definitely in front of the other. In order to tell which side of the line the eye position is on, the line connecting the two points is made into a vector, oriented such that it has a negative Z component, referenced to the coupling-test coordinate system. Figure 2-12 illustrates two situations where shapes are coupled, and shows the appropriate coupling vector for each situation.

The code below assumes that coupling is indicated by a triple (X_1 X_2 S), where X_1 and X_2 define the coupling vector and S points to the other shape. This code sets up the triple for each shape such that the coupling vector has a negative Z component.

```
/* let A = X_max of S_1 , B = X_min of S_2 (coupling-test coordinate system) */

if A_z > B_z then
    couple_1 = (A^-1 B^-1 S_2)
    couple_2 = (B^-1 A^-1 S_1)
else
    couple_1 = (B^-1 A^-1 S_2)
    couple_2 = (A^-1 B^-1 S_1)
endif
```

A^{-1} and B^{-1} represent the points transformed back to the object space coordinate

system. Thus, the output from the test for coupling is a pair of triples ($couple_1$ and $couple_2$) to be associated with S_1 and S_2 , respectively. Each triple (P_1 P_2 S) contains the vector P_1 P_2 and a pointer to the other shape.

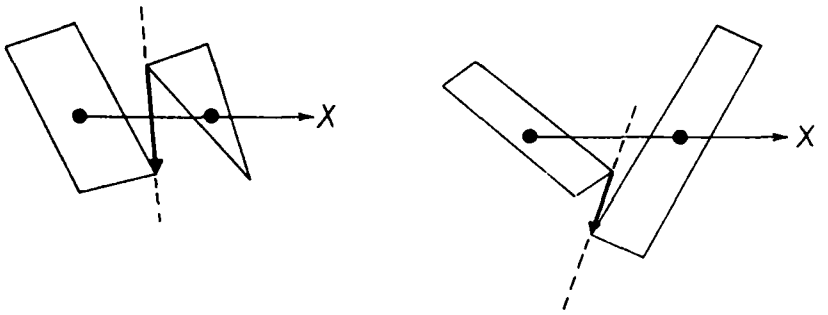


Figure 2-12

Coupling Vectors

Decoupling of two shapes is done as a shape is encountered in the octal-tree traversal. When shape S_1 is processed, decoupling must first take place to insure that no shapes have priority over S_1 . Upon encountering the coupling triple for S_2 , a test must determine the relative priority of S_1 with respect to S_2 . Assuming a coupling triple (P_1 P_2 S_2), S_1 is behind S_2 if the vector P_1 P_2 has a negative X component (eye coordinate system). Otherwise, S_1 is in front of S_2 . The priority of S_1 with respect to S_2 depends on whether the objects in the scene are being drawn back-to-front or front-to-back. The *Behind?* function below illustrates the simple decoupling test.

```
/* Returns true if shape # 2 is between shape # 1 and the eye position. */

boolean function Behind? (triple)
    P1 = triple.p1
    P2 = triple.p2
    return (P2.x < P1.x)
end Behind?
```

This chapter has dealt with object space decomposition as a way to establish a priority order among objects in a scene. The object space is decomposed until each centroid occupies a unique sub-volume of the space, and a traversal of the octal-tree establishes the ordering. Basing the decomposition on centroids can result in anomalies in the priority order, so a method of coupling/decoupling objects is used. Centroids can be chosen so as to minimize the coupling for the objects in a scene. This results in a method of traversing the octal-tree, with possible "detours" for the decoupling of shapes.

Chapter 3. Hidden Surface/Line Determination

Hidden surface removal involves the establishment of priority among objects in a scene, to determine which objects hide other objects, along with the calculation of visible portions of the scene. Chapter Two dealt with a method for establishing a priority ordering for objects based upon object space decomposition; Chapter Three concentrates exclusively on algorithms for calculating the visible portions of a scene, given a priority ordering for the objects in the scene. Three algorithms are presented in detail. The purpose for exploring each one is to evaluate the algorithms from a parallel processing viewpoint. Compared with sequential processing, parallel processing could either perform similar work in a shorter time (removing hidden lines faster), or perform more work in the same time (adding shading and/or shadows to an existing capability).

The first algorithm, intended for use on raster scan displays, conceptually uses two quad-trees [7, 9, 10, 11], one to represent the display area and the other to represent a polygon to be drawn. Binary subdivision is used to partition the display area into quadrants which are either completely full or completely empty. By merging the two quad-trees correctly, a polygon may be added to the display such that the hidden areas of the polygon are not displayed. The second algorithm is purely geometric. With the

aid of a priority list, the polygons are tested against each other for intersecting edges in order to determine visible line segments, which are then displayed. While being rather brute force, the algorithm offers a considerable amount of concurrency. These two algorithms both have problems in complexity growth and/or response time, so the third algorithm combines features of the two previous algorithms in an attempt to alleviate those problems. This third algorithm uses a quad-tree to divide the display into manageable quadrants, each of which is either empty or populated with polygons. Polygons are physically subdivided at the quadrant boundaries and the priority order of the original polygons is maintained. Each quadrant is processed in parallel using the geometric algorithm. A complexity analysis for both time and computation is presented for each algorithm.

Test results from studies run on a parallel processing simulator [8] are also presented. The simulator uses a simple instruction count as the measure of computational work, and is able to keep track of a number of parallel tasks. These tasks exhibit a hierarchical ancestry. Two terms used in the discussion of the test results are important: *critical path* and *parallelism factor.* The *critical path* is the longest path in the execution hierarchy (measured in the number of instructions), and can be viewed as the fastest response time for the task, assuming no limit on the number of tasks that can execute concurrently. The *parallelism factor* is the average number of tasks that are

active, and is computed by dividing the total number of instructions by the critical path. This gives a measure of the speedup that could be obtained by using a truly parallel machine, relative to a sequential machine executing the same algorithm. Further discussion on the simulator may be found in Appendix A.

3.1 Quad-Tree Algorithm

3.1.1 Raster Scan Displays

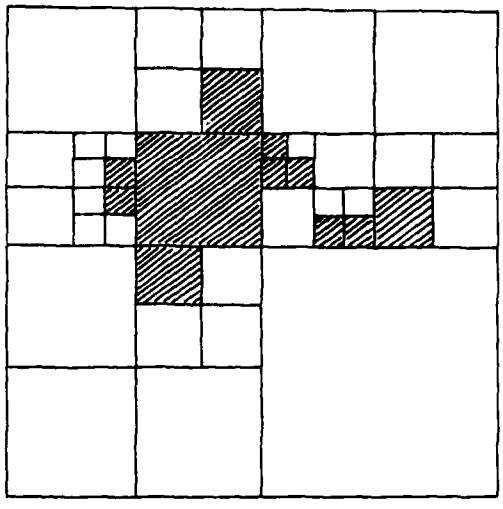
Raster scan and bit map displays can be used as line drawing displays, but more often they are used to cover entire polygonal areas rather than simply outline them. Among other things, this allows for the continuous shading of objects and the use of colors to distinguish objects on the screen. Using such a display, a "correct" picture is easily generated if the priority ordering of the polygons is known. This is accomplished using a *painters' algorithm* [16], in which the polygons are displayed in reverse order, from back to front. Polygons which should appear to be nearer to the viewer do so because they are drawn over earlier polygons. The algorithm involves simple scan conversion of the polygons, and any scene can be rendered in linear time with respect to the number of objects in the scene. It is used in this thesis as a basis for comparison with the techniques which follow.

3.1.2 Quad-Trees

A quad-tree is a tree whose leaves represent areas of a picture; each node of the tree corresponds to a unique area of the picture (Figure 3-1). A picture is considered to be a square array of colored pixels, and each leaf of the quad-tree is labelled with the color of the region to which it corresponds. By definition, no parent node may have all its descendant leaves the same color, since the tree could be more compactly represented by eliminating the descendants and coloring the parent. For pictures which possess large contiguous areas of the same color, a quad-tree can offer considerable storage savings over a two-dimensional array (a typical display memory, for instance), because each leaf represents a block of color usually much larger than a single pixel [11].

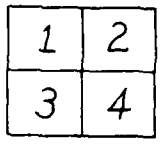
3.1.3 The Algorithm

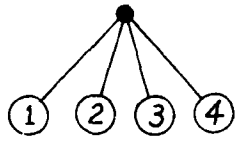
The algorithm presented below conceptually uses two quad-trees to accomplish hidden surface removal. One tree represents the display area and identifies those areas of the display which are "occupied" by polygons. The other tree represents a polygon. The roots of both trees correspond to the entire display area. By combining the trees, the polygon may effectively be added to the display, and since the screen quad-tree



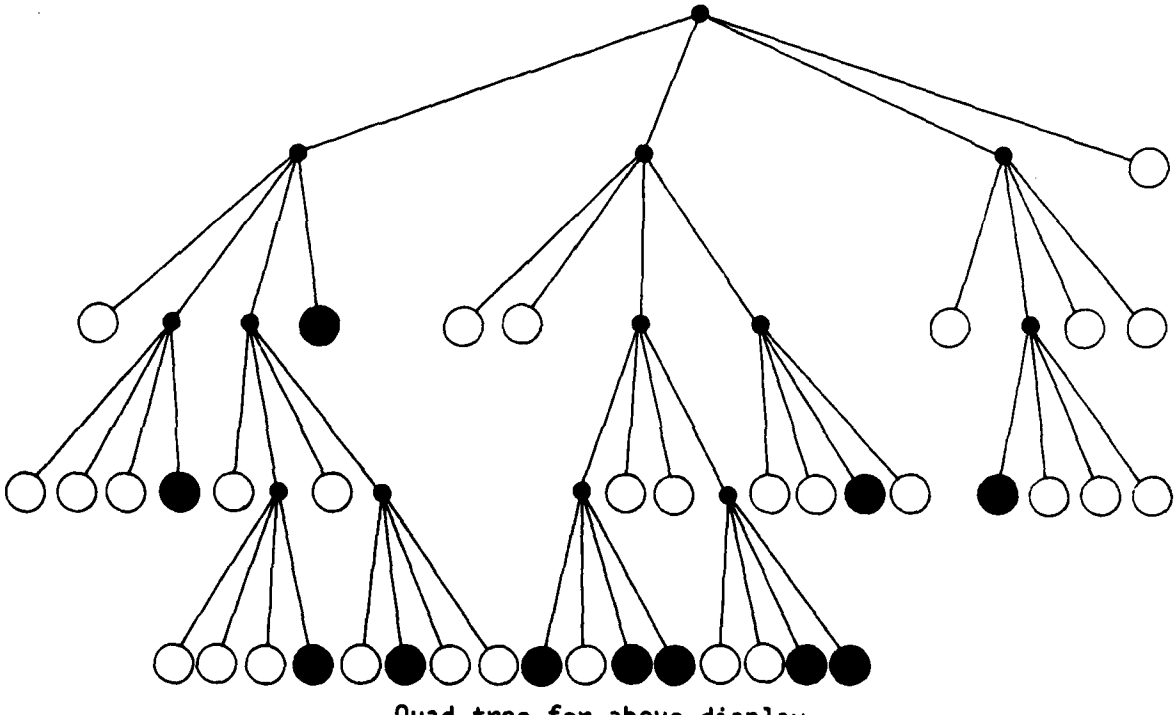
display

occupied quadrant





quadrant numbering



Quad-tree for above display

Figure 3-1

contains the information about the state of the display, the merging of the quad-trees may be done so that the polygon does not obscure any part of the existing display. The polygons in a scene are processed in priority order, and as they are displayed, the areas they occupy are added to the already occupied areas of the screen through the merging process. The conversion of each new polygon to a quad-tree, followed by the merging of that tree with the one representing the display screen (with the latter taking priority), thus provides a technique for accomplishing hidden surface removal. Notice that if the quad-tree for the new polygon took priority, we would essentially have the painter's algorithm. However, because the screen quad-tree takes priority over the quad-tree for the new polygon, the polygon can only appear in areas which are unoccupied at the time of merging. Thus, a front-to-back display protocol is assumed for this algorithm.

The discussion which follows centers on methods for converting an arbitrary convex polygon to a quad-tree, and on the process of merging two quad-trees to accomplish the hidden-surface removal. This separation of functions (creation/merging) is not really necessary. In fact, the implementation of the algorithm as two functions is inefficient since it forces a sequentiality that is not inherent to the algorithm. Presented last is a version of this algorithm which combines the two functions into one which traverses/mutates the screen quad-tree as dictated by the polygon to be displayed.

3.1.3.1 Converting Polygons to Quad-Trees

The reason for using a quad-tree to represent individual polygons as well as the display area is to save storage space and processing time by representing relatively large areas of the screen by a single node in the tree; the size of the area is implied by the level of the node within the tree, and the location of the area is implied by the position of the node (Chapter 2). Converting a polygon to a quad-tree creates a tree of maximum depth q, where q is the base two logarithm of the screen size (assuming a square display area). A quad-tree thus maps the input polygon to the display screen. Hunter [9] accomplishes this conversion by combining an outlining algorithm with a coloring algorithm. The outlining algorithm creates what Hunter terms a "roped" quad-tree whose leaves define the perimeter of the polygon. The border leaves are tied together using pointers so that the perimeter may be followed. The coloring algorithm takes as input a roped quad-tree and returns a quad-tree which represents a solid-color polygon. The time required to outline and color a polygon is $O(v+p+q)$, where v is the number of polygonal vertices, p is the largest integer not greater than the perimeter of the polygon, and q is the maximum depth of the quad-tree.

In this thesis, we use a polygon conversion process which evaluates smaller and smaller regions of the screen until each region can be a single color (an implementation

is presented in Appendix B). Binary decomposition, in the fashion of the Warnock algorithm, determines which areas of the screen are evaluated. Because each node of the quad-tree corresponds to a unique region of the screen, there are no data dependencies between sibling nodes (nodes on the same level of the tree); this implies a possibility for processing each quadrant of the screen in parallel. The parallelism generated will resemble the resultant quad-tree.

The algorithm recognizes only three possibilities for any quadrant: completely full, completely empty, or *unknown*. Leaves in the quad-tree exist only for the first two cases, and the quadrant must be subdivided for the third. In subsequent discussion the terms full/occupied/black and empty/unoccupied/white are used interchangeably. A black quadrant can exist only if the polygon completely surrounds the quadrant, while a white quadrant exists only if there is no intersection between the areas of the polygon and quadrant. Any other situation is termed unknown and requires subdivision of the quadrant. Figure 3-2 illustrates several polygons in relation to a quadrant, along with the appropriate "label" for the relation. Subdivision proceeds only as far as the single pixel level.

The conceptual simplicity of allowing only "black or white" quadrants belies the complex calculations required to make the determination. Simulation results for the

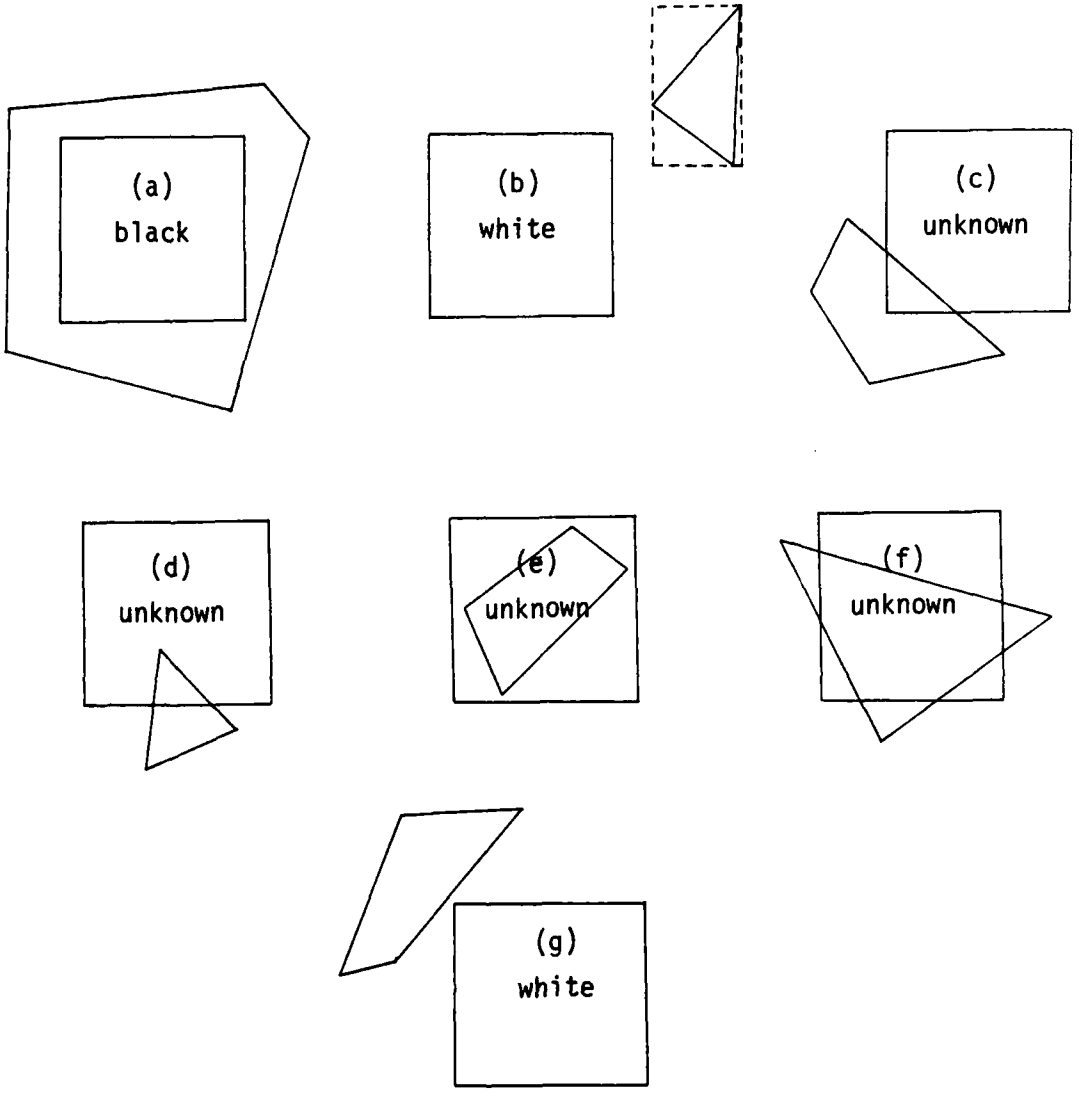

(a)
black
(b)
white
(c)
unknown
(d)
unknown
(e)
unknown
(f)
unknown
(g)
white

Figure 3-2

quad-tree algorithm will be presented, and it is important to note that the majority of work done by the algorithm aries from this black/white determination. The discussion which follows is geared towards the implementation used in that simulation study.

Assuming a convex polygon, a quadrant can be "black" if and only if each corner of the quadrant is inside the polygon. This is the same as saying that the polygon "surrounds" the quadrant. An important computation for determining a "black" quadrant tests if a point (x,y) is contained within the boundaries of a polygon. Given a point known to be inside the polygon (see below), the test point is inside the polygon if and only if for each line in the polygon, the test point and the known inside point are on the same side of the line. This test is accomplished by plugging the coordinates of both points into the formula $Ax + By + C$ (which is the left hand side of the line equation). The points are on the same side of the line if the resultant values have the same sign. Calculating a point known to be inside a convex polygon is a straightforward calculation, as shown in Figure 3-3. Given two edges of a polygon which share a vertex, it is simply the midpoint of the line joining the midpoints of the edges. This point need be computed only once, and can be stored as part of the polygon description. Determining whether a quadrant is surrounded by a polygon can be done in $O(v)$ time, v being the number of polygonal vertices.

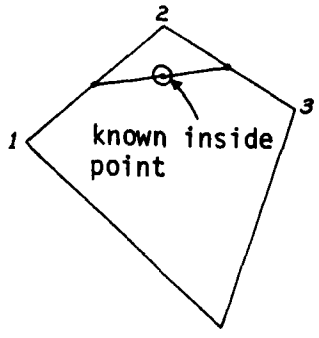


Figure 3-3

Calculation of a Point Known to be Inside a Polygon

If a quadrant is "white," it is because the polygon and quadrant are disjoint; that is, there is no intersection between the two areas. However, there is a distinction between "simple" and "complex" disjoint situations. The simple case can be easily determined with a "min-max" test between the quadrant and the minimum enclosing rectangle for the polygon. Figure 3-2b illustrates such a situation. Assuming that the description of that enclosing rectangle is stored as part of the polygonal information, there is an upper bound of four comparisons necessary to determine the simple disjoint case. The complex situation requires considerably more computation, and can be described as occurring when: (1) all polygonal vertices lie outside the quadrant, and (2) no intersections exist between the polygonal edges and the quadrant boundaries. See Figure 3-2g for an example. This computation is expensive because there is an upper

bound of O(v) simultaneous linear equations which must be solved to verify an absence of boundary intersection.

Because of the complex computations necessary to determine a complex disjoint situation, several intermediate situations are recognized, each of which are "unknown" and require subdivision. This provision for intermediate cases is done because it is desirable to find out as soon as possible whether subdivision of the quadrant is necessary, and early determination of an "unknown" situation can avoid unnecessary computation. Such situations are illustrated in Figures 3-2(c-f) and are presented in order of increasing computational requirements. Assuming the quadrant is not "black" and not a simple case of "white," then if any quadrant corner is inside the polygon's boundaries (Figure 3-2c), the situation is "unknown." This case is available at no extra cost since each corner is tested anyway to determine a "black" situation. Next, if any polygonal vertex is within the quadrant boundaries (a simple computation), then subdivision is also dictated. Figures 3-2(d-e) illustrate two likely situations. Finally, it is possible for all vertices to be outside the quadrant and all quadrant corners to be outside the polygon, as shown in Figures 3-2(f-g). Here, intersections between the boundaries distinguish between an unknown situation (Figure 3-2f) and a complex "white" situation (Figure 3-2g).

3.1.3.2 Merging Two Quad-Trees

Given a quad-tree which describes the current state of the display area, and another quad-tree which describes the new polygon, the two trees must be merged to form a new quad-tree (in essence, this adds the new polygon to the display area.) It is important to realize the different roles played by these two quad-trees. For the screen quad-tree, there is no distinction between individual polygons; nodes are either "occupied" by some previous polygon, or "empty" and available for display. The quad-tree for the new polygon identifies those areas of the screen which the polygon would occupy if alone on the screen, plus the color of the polygon (or the grey scale for a monochrome display). As shown in Figure 3-4, the discovery of a "full" quadrant for the new polygon might result in the consolidation of some quad-tree children nodes, but the actual display will not reflect this result; the new polygon may appear only in empty areas of the screen.

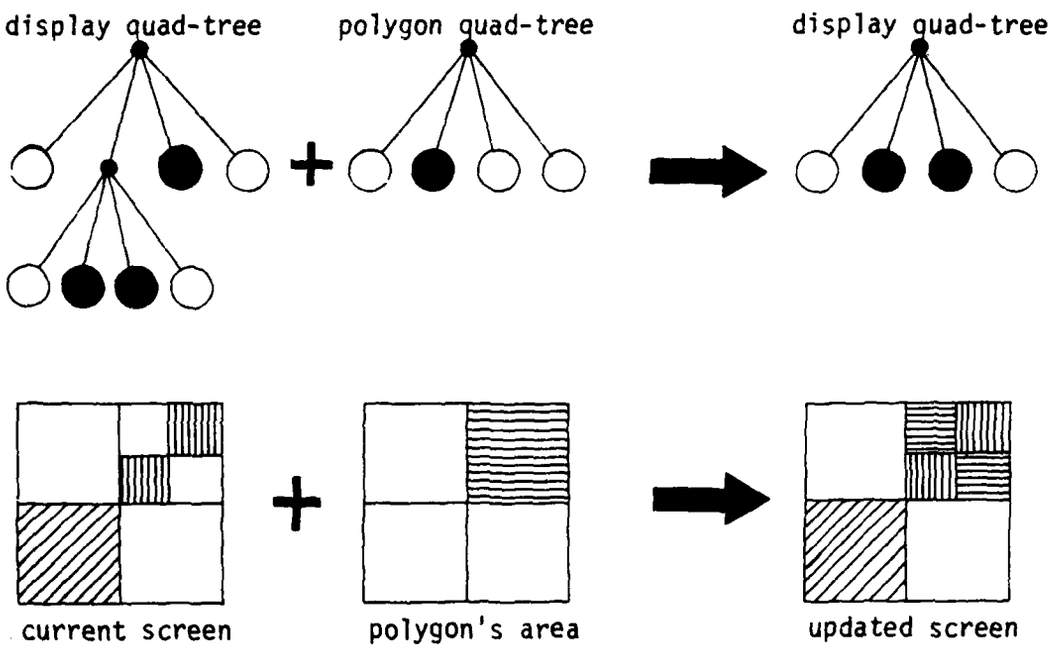


Figure 3-4

Addition of Polygon to Display (Different Effects on Quad-Tree and Screen)

To merge the screen quad-tree and the new polygon's quad-tree, the two trees are traversed in parallel, and the corresponding nodes of each tree are compared. Given an "old" node from the screen tree and a "new" node from the new polygon's tree, the action depends on whether the old node is a leaf:

If the old node is a leaf:

(a) If the old node is empty, the new node (whether leaf or sub-tree) takes its place. The occupied leaves of the new node may be displayed on

the screen.

(b) If the old node is occupied, nothing happens (it does not matter what the new node contains).

Otherwise, if the old node is not a leaf:

(a) If the new node is empty, the old node remains intact.

(b) If the new node is occupied, then it will take the place of the old node (since every empty sub-node in the old node will be filled in). However, for display purposes, only the screen quadrants corresponding to the empty leaves of the old node may be filled in with the color of the new node. This is because the new polygon must not "paint over" old areas of the display.

(c) Otherwise, both nodes are themselves trees, and the algorithm recurses to process each node of the trees in parallel.

3.1.3.3 Combining Tree Creation with Merging

It is actually not necessary to separate the function of creating a quad-tree for each new polygon from the function of merging a new quad-tree with the tree for the screen. This separation imposes a sequential nature on the processing which is really not inherent to the algorithm. Because both quad-trees are isomorphic (the root node

of each tree corresponds to the total screen area), it is possible to combine the two functions. This results in a traversal of the screen quad-tree, such that at each level of the tree, decisions are made that determine whether the tree is modified, left intact, or traversed further (because a decision cannot be made). Modification of the tree should be viewed as causing an update of the display.

This version of the quad-tree algorithm combines the tree-creation and tree-merging functions by recursively processing nodes in the screen quad-tree. This discussion distinguishes between the "quadrants" of the display area and the "nodes" of the quad-tree which represent the display. The quad-tree is traversed and mutated depending on the current state of each node and the situation between the polygon and the quadrant to which that node corresponds:

If the node is a leaf and "occupied," then nothing happens since no part of the polygon could appear in the corresponding quadrant.

If the node is a leaf and "unoccupied," then that quadrant is available to the current polygon. Here, the algorithm tests to see whether the quadrant is "black or white"; that is, whether the quadrant either lies completely inside or outside the polygon.

(a) If the quadrant is "black," the quadrant is painted with the color of the polygon and the node is labelled as "occupied."

(b) If the quadrant is "white," the corresponding tree node remains "unoccupied."

(c) Otherwise, the quadrant situation is "unknown," so the algorithm subdivides the node into four unoccupied nodes and then recurses for the subquadrants.

Otherwise, the quad-tree node already is subdivided. The "black or white" situation is still important:

(a) If the quadrant is "black," the **unpainted** areas of the quadrant are painted with the color of the polygon and the quad-tree node is labelled as "occupied."

(b) If the quadrant is "white," the quad-tree node remains intact.

(c) Otherwise, the algorithm returns a node which is a result of processing the four sub-quadrants of the quadrant in parallel.

3.1.4 Test Results

In order to test the parallelism available in the combination version of the quad-tree algorithm, a test case was constructed from twenty randomly placed polygons. The test scene was processed on the LCODE simulator (see Appendix A) to

gather statistics on (1) the overall efficiency of the algorithm, and (2) the parallelism available in the algorithm. As with EPA mileage estimates for automobiles, these statistics are useful only for comparison with the other algorithms presented in this thesis.

The test scene was first evaluated using a simple painter's algorithm which scan-converted each polygon in the scene. The screen was assumed to be 256 by 256 pixels. This method was strictly sequential and was used as a means of comparison. Using the LCODE simulator as a processing yardstick, the painter's algorithm for the twenty polygons required 64,079 instructions. This number could be decreased by scan-converting each polygon in parallel, saving the pixel information, then sending that information to the display memory in the proper sequence. The time to process a scene using the painter's algorithm grows linearly as a function of the number of objects in the scene.

The numbers for the quad-tree method were substantially higher, even accounting for parallelism. The total number of instructions executed in processing the scene were 8,638,688 with a critical path of 176,455. This gave a figure of approximately 49 for the parallelism factor. Thus, even with the parallel processing capability, the quad-tree method required almost three times the processing time of the

painter's algorithm.

Over the course of the testing, it was observed that the implementation of the algorithm was deficient in its management of the quad-tree. As the nodes in the tree were filled, no attempt was made to consolidate the children of any particular parent node. This meant that if a parent node had four children leaf nodes that were occupied, then the algorithm failed to eliminate the children and label the parent node as occupied. The algorithm was modified to perform a test as recursion occurs, such that if the four nodes of any sub-tree are occupied, then an occupied node (instead of the sub-tree) is returned. Processing the scene with the modified algorithm produced some interesting results. The total number of instructions decreased to 7,930,071 (a decrease of eight percent); this meant that the algorithm did not have to recurse as far to determine whether a node was occupied or empty. However, the critical path increased to 218,592 (an increase of 24 percent). Thus, the modified algorithm provides a slower response time even though the total work done by the algorithm decreases. When no consolidation took place, the quad-tree for the test scene grew to a size of 889 leaves. The modification resulted in a tree with 568 leaves, and somewhat less than 20 percent of the attempts at consolidation were successful.

3.1.5 Computational Complexity

The quad-tree approach requires a great deal of computational work, being roughly a function of the perimeter of the total screen area taken up by the polygons in the scene, *at each point in the processing sequence.* At first glance, it might appear that the algorithm itself is doomed to relative failure because of the recursion required to produce a quad-tree down to the single pixel level. Arbitrary polygons have edges which are predominantly slanted (as opposed to vertical or horizontal), meaning that the algorithm must recurse to the single pixel level in order to process the perimeter of the polygons. Thus, in effect, the quad-tree method emulates a Digital Difference Analyzer (DDA) for processing the edges of the polygons. This could hardly be less efficient; the algorithm streaks along when it finds large blocks of the screen which are either empty or completely full, but processing the edges results in enormous amounts of recursion. Significant testing is done at each level of recursion to see if a given node should be either full or empty, and this processing time is completely wasted in the event that more recursion is deemed necessary.

For the first few polygons in a scene, assumed to be spread across the screen in a fairly random fashion, the quad tree will be at its maximum depth and fairly complex, with the number of unit pixel leaves proportional to the total perimeter of the polygons.

As more polygons are processed, the decomposed nature of the quad-tree will force more recursion in order to access the information at those leaves. This will be in evidence when two polygons overlap, and the algorithm has to fill in the area of the later polygon without "crossing the lines" of any polygons drawn earlier. For these situations, recursion will again occur down to the single pixel level, but when the additional polygonal area is added to the quad-tree, it may be possible to consolidate some of the leaves such that if all four children in a node are occupied, then the parent can be marked as occupied and the storage space for the children reclaimed. Such consolidation not only decreases the storage requirements for the quad-tree, but it means that for all future polygons, recursion for an occupied quadrant goes no deeper than the level for that quadrant. This implies that the processing requirements for a scene will be considerable until such time that the screen area begins to fill up. There will generally be a point in the processing when the tree ceases to grow and begins to consolidate; at this point, the processing required to process additional polygons will begin to decrease. At some hypothetical point, the screen may become so full (90-95%) that processing additional polygons becomes considerably less expensive.

Of course, the number of polygons required to fill up a screen area is largely dependent on the nature of the scene being depicted. A close-up view of a cityscape, for example, may require relatively few polygons, since one or more of the polygons

may be close enough to the eye position to completely fill the screen. That same city, when viewed from an eye position much farther away (from outer space, say, as opposed to on the doorstep of one of the buildings), may occupy only a small portion of the screen area. In such a case, there would be a large number of polygons lying "on top" of one another or nearly so. It was decided to gather statistical data on the growth of the algorithm's complexity, and to assume a pseudo-worst-case scenario of using relatively small polygons and randomly placing them around the screen. The methodology was to take ten arbitrary polygons, each less than one percent of the total screen area, and for each iteration of the process, randomly pick one of the shapes and randomly place it within the screen area (some were placed partly off the screen so as to simulate a need for clipping), until such time as a desired percentage of the screen area was occupied. Because of overlap between the polygons, the number of polygons required to fill the screen area to the desired degree grew as shown in Table 3-1 and the corresponding graph.

% full	# polygons
10	30
20	68
30	117
40	163
50	224
60	337
70	513
80	1136
90	2720

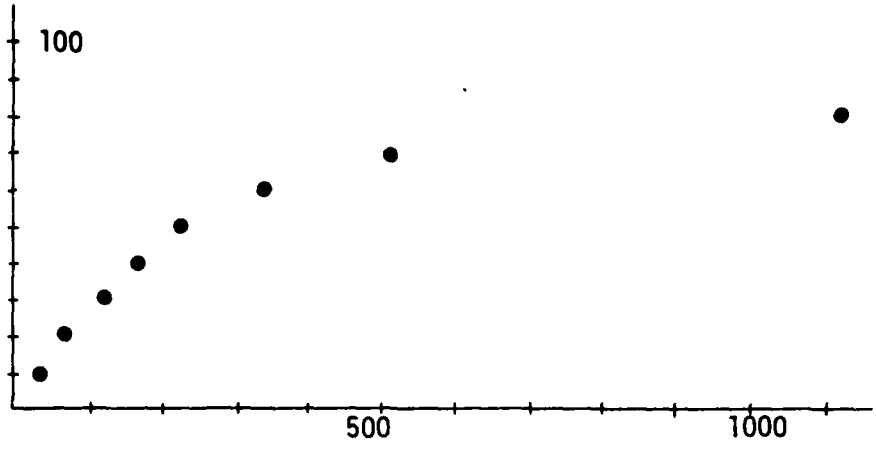


Table 3-1
Polygons Required to Fill Screen to Given Percentage
(Average of 10 Runs)

The next step in the process involved creating a quad-tree for each state of the screen (nine quad-trees corresponding to 10% - 90% full). The random placement of the polygons meant that "full" quad-tree nodes of various size would be randomly distributed about the tree. It was then desired to know, for a screen that is full to a given degree, how much processing time the next polygon would require. For each of the nine quad-trees, statistics were collected on several randomly-selected, randomly-placed polygons. The results are tabulated and displayed in Table 3-2 and the associated graph.

% full	work
0	102
10	100
20	92
30	87
40	82
50	76
60	69
70	62
80	46
90	33

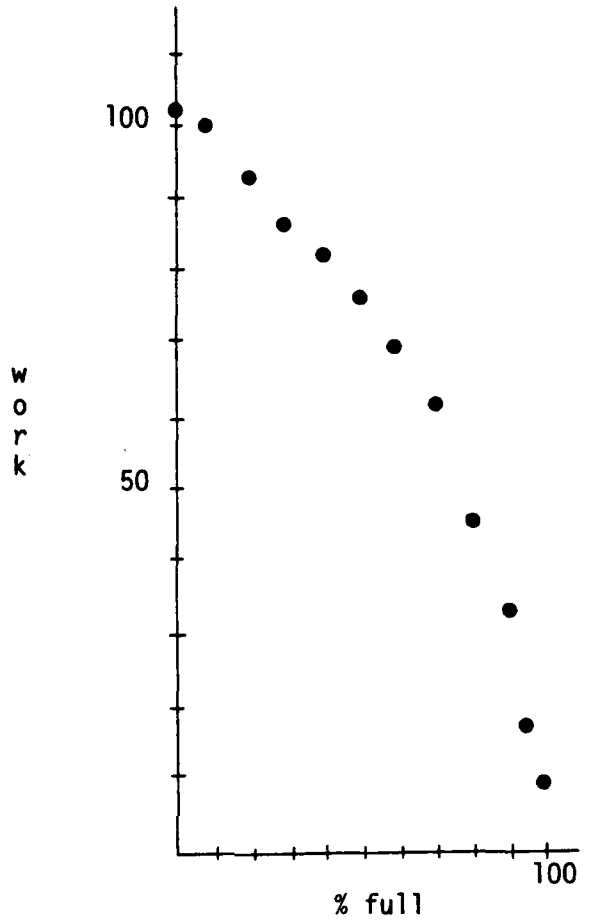


Table 3-2
Work Necessary to Process Polygon, Given Quad-Tree Full to Given Percentage
(Average of 100 Polygons)

These two pieces of information constitute a means for calculating the amount of computation required to process polygon *n*, given that *n-1* polygons have already been processed. Using *n-1*, we "lookup" the degree to which the screen area will be filled, and for that degree of fullness, we lookup the amount of processing that the next polygon will typically require. Calculating the complexity growth for the algorithm then involves "integrating" the information under the curves by summing the

processing requirements for each of *n* polygons and graphing the values. The resultant curve is shown in Figure 3-5, and illustrates the rapid growth characteristics of the algorithm for the early stages, followed by the levelling of the curve as the number of polygons approaches 10000.

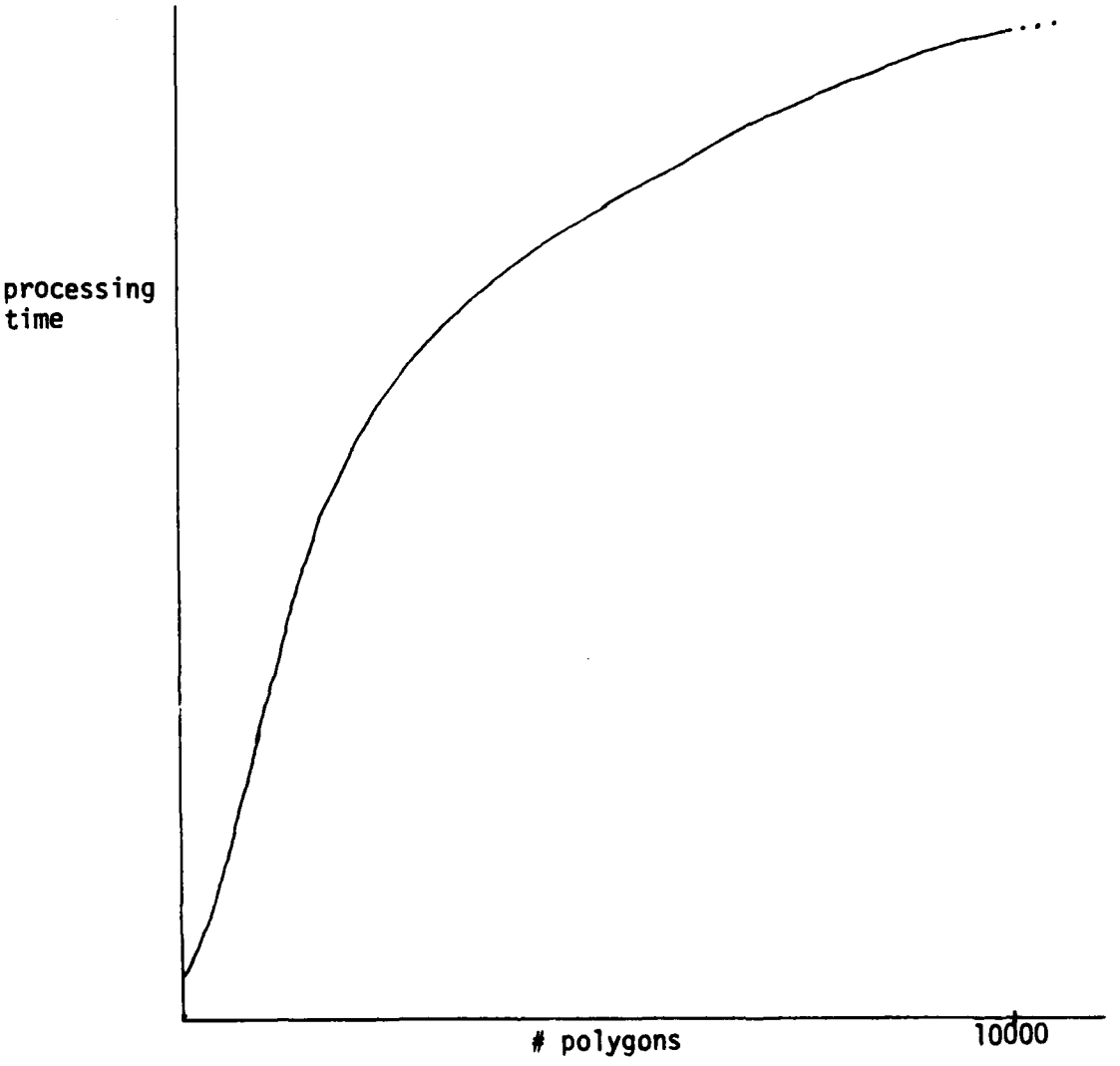


Figure 3-5

Growth in Processing Time as a Function of the Number of Polygons

These statistics provide a non-rigorous estimate of the complexity growth for the quad-tree method, and show that the algorithm would most likely not be suited for a rapid depiction of a very simple scene containing only a small number of polygons. However, we should point out that we are not interested in such applications, for there are quite a few methods suitable for such scenes. One consideration which offers improvement to this algorithm would be the creation of a number of independent processes at such time as the screen became full to a certain degree (probably above 75%). These processes would pre-process the list of remaining polygons, filtering out any which would be completely hidden by any node in the quad-tree. A new process could be started each time consolidation of nodes occurred, since that would indicate that a larger quadrant had become completely occupied. For a large number of polygons (greater than 2000) and a substantially-full screen, this could mean that the computational growth could approach zero; that is, the processing of 10000 polygons might require no more processing (real) time than 9000 polygons, since the filtering processes would be able to eliminate a large number of polygons before the main algorithm had to look at them. This twist to the quad-tree approach smacks of the Warnock algorithm, since the list of polygons to be processed could be prematurely terminated by the discovery of a quadrant which was full ("surrounder" polygon).

3.2 Geometric Algorithm

3.2.1 Line Drawing Displays

For line drawing displays, the second phase of hidden surface removal becomes one of determining visible line segments, rather than visible polygonal areas. These displays, such as stroke refresh and storage tube CRTs, are very inefficient at coloring entire regions of the screen; because of this, only the edges of the polygons are drawn. Unlike the painters' algorithm for raster scan devices, a line drawn on a refresh or storage device will remain visible until explicitly erased. Any algorithm for a line drawing display must take great pains to draw only those portions of polygonal edges which will be ultimately visible to the viewer.

3.2.2 The Algorithm

The approach taken by this algorithm is a series of geometric calculations that determine all points of intersection between the edges of all polygons in the scene. With these points of intersection and a priority order for the polygons (derived from the octal-tree traversal of Chapter Two, for instance), visible line segments may be constructed and displayed. A great deal of concurrency exists in this algorithm, and in the discussions below we attempt to take advantage of all available parallelism.

The algorithm focuses solely on convex polygons, each of which is defined by three or more edges. An edge is defined by two endpoints. The term "old polygon(s)" is used to describe polygons which have previously been processed. They have a higher priority than any polygons yet to be displayed, and thus a "new polygon" is not allowed to obscure any portion of the existing display. This constrant implements a front-to-back protocol for the polygons. An "invisible line segment" is a segment of a polygonal edge which is obscured by a polygon of higher priority. For any given edge, then, the invisible line segments caused by all higher-priority polygons can be masked onto the original edge, and any remaining portions of the edge are completely visible.

Given a "new" polygon and a number of higher-priority polygons, the algorithm calculates which edges (or portions of edges) of the new polygon are visible with respect to the other polygons. This calculation involves a number of tests between the new polygon and each "old" polygon to see if the edges of the new polygon and the old polygon intersect, or if one or more of the new polygon's vertices lie within the boundaries of the old polygon. If so, then for each edge in the new polygon, all *invisible* line segments are determined based upon the points of intersection. Figure 3-6 illustrates the four general relationships which a polygonal edge might have with a higher-priority polygon: (a) both endpoints are visible, but the line is trisected; (b) one endpoint is visible, and the line is bisected; (c) both endpoints are invisible (thus for

convex polygons, the entire line is invisible); and (d) the line and the polygon are disjoint. The points defining the invisible segments consist of either original endpoints or points where edges intersect. A min-max test is useful for rapidly eliminating polygons which cannot possibly obscure the polygon under consideration.

For each line in each successive polygon according to the priority order, the invisible line segments resulting from the tests with the higher-priority polygons are used to mask the original edge until the line completely disappears or until all invisible line segments have been processed. Processing a large number of line segments can be made more efficient in some cases by having, for each line of the new polygon, a flag which marks the condition that the line is completely hidden by any polygon of higher priority. In such cases, no further processing need be done for that particular edge of the new polygon. In either case, for each line segment, there will ultimately exist zero or more line segments which are visible; these can be sent to the display device for plotting.

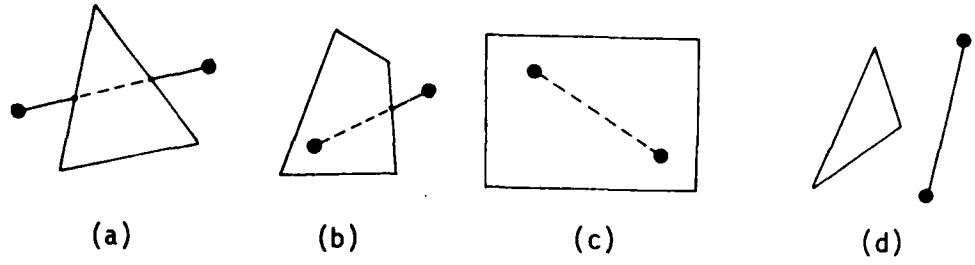


Figure 3-6

Possible Relationships Between an Edge and a Polygon

3.2.3 Complexity Analysis

This algorithm, like many algorithms that are conceptually simple, is rather brute force in nature. Assuming the correct priority ordering output by the first phase of the hidden surface algorithm, this geometrical method exhibits the $O(N^2)$ complexity growth of other algorithms such as Roberts [18]. Each polygon is compared only with those polygons which precede it (have higher priority); thus, for the Kth polygon, K-1 polygons must be compared. This means that the total number of comparisons for a scene with N polygons would be $[N(N-1)]/2$ or $O(N^2)$. However, due to the algorithm's simplicity, there are not many inherent data dependencies. As a result, almost all computations may proceed concurrently. This is done by taking a list of all polygons in order of ascending priority. By walking down the list, a number of tasks

may be forked to execute in parallel. Each sub-list (all elements in the previous list except the first) contains the information necessary for the comparisons to be made. The first element of each list points to the new polygon, while the remaining elements represent previous polygons, each of which has a priority higher than the new one. For example, consider five polygons in a scene ordered as (P2 P4 P1 P3 P5) where priority increases from left to right. Five tasks are forked off:

```
Process (P2 P4 P1 P3 P5) -- P2 compared to P4, P1, P3, P5

Process (P4 P1 P3 P5) -- P4 compared to P1, P3, P5

Process (P1 P3 P5) -- P1 compared to P3, P5

Process (P3 P5) -- P3 compared to P5

Process (P5) -- P5 drawn automatically
```

Thus, given a prioritized list:

(1) Each polygon in the list may be processed in parallel, since the information necessary to process each polygon is contained in the remainder of the list.

(2) Within the list of higher-priority polygons, each process to calculate the invisible line segments may again be conducted in parallel.

(3) The calculation of line intersections for each pair of lines may be done concurrently.

Assuming the realization of all available parallelism, the time required to process a scene using the geometric method becomes linear, being the time required to walk down the prioritized list of polygons and fork off the necessary tasks. The real work performed by the algorithm, comparing a new polygon against an existing polygon, can be done in constant time O(v), where *v* is the number of vertices in a polygon. But while the lack of data dependencies implies no significant growth in the critical path of the algorithm, the computational growth implies a requirement for $O(N^2)$ growth in the number of processors in order to achieve full parallelism. One could seemingly improve the complexity of the algorithm by having a tree-like priority list; this would enable the algorithm to skip some clearly irrelevant polygons. As will be seen shortly, the algorithm in Section 3.3 does just this.

3.2.4 Test Results

The results obtained from this purely geometrical approach were interesting. Processing the twenty polygons in the test scene required a total of 772,435 instructions with a critical path of 12,158; this gives a parallelism factor of close to 64. Compared to

the painter's algorithm, the geometric approach required roughly ten times as much total work, but took only one fifth as long. Besides the relative lack of recursion in this approach, the other reason for better results seems to be the use of the FORK mechanism (see Appendix A) to create the parallelism (PCONS is still utilized indirectly through the PMAPCAR function). The FORK mechanism allows for the generation of parallel tasks without the overhead required of the PCONS mechanism (joining the two values together in spite of which task finishes first). FORK is used to start a parallel task for each polygon in the scene by CDRing down a list of the polygons in the scene (as discussed above).

Additional efficiency is obtained because the tasks are forked in order of descending processing requirements. The last task started requires the least processing time, since the first polygon is drawn immediately. The number two polygon only needs to compare against the first polygon, and so on. This means that the first task started (the most time-consuming) is well on its way to completing as the last task is forked, making the most effective use of the forking mechanism, in light of the processing requirements.

3.3 Geometric and Quad-Tree Combination Algorithm

The final algorithm for this thesis involves a combination of the quad-tree algorithm of Section 3.1 and the geometric algorithm of Section 3.2. The major problem with the quad-tree algorithm is its tendency to create a large workload by requiring a sub-quadrant to be a single color; in many cases this causes recursion down to the maximum resolution of the display. The geometric algorithm exhibits large computational growth characteristics, but is very fast for a reasonably small number of polygons. Conceptually, it is possible to modify the quad-tree algorithm so that a quadrant need not be empty, full, or subdivided, but may be empty, *populated*, or subdivided. Being populated means that a quadrant has one or more polygons within its boundaries. Using this relaxation of the quad-tree constraints, then, the geometric algorithm is used to solve the hidden line problem within each quadrant. Details on quad-trees may be found in Section 3.1.1. This section will present a detailed discussion of this combination algorithm, followed by simulation test results and a computational complexity analysis.

3.3.1 The Algorithm

As with the previous two algorithms, this one assumes as input a prioritized list of polygons. The algorithm processes each polygon in order, using the geometric method (Section 3.2) to determine the visible line segments. Initially, the quad-tree will contain only the root node, corresponding to the entire display area. Polygons will be processed and added to the list of polygons for that node. Because of the $O(N^2)$ growth in computation in the geometric method, however, the algorithm imposes an arbitrary limit to the number of polygons which may occupy the node. When the population for the root node exceeds the limit, the screen is logically subdivided into four quadrants, the root node is divided into four sub-quadrants, and the existing polygons are physically divided among those quadrants (Figure 3-7). This should result in each quadrant having a population of polygons less than the maximum allowed, unless the polygons are all stacked one on top of the other.

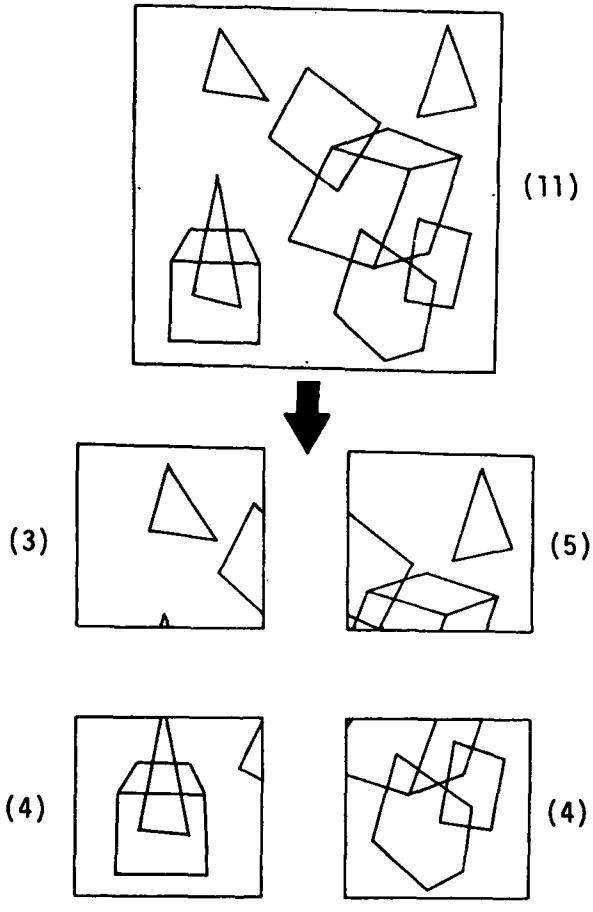


Figure 3-7

Quadrant Population Decreasing Upon Subdivision

One could view this as a process in which polygons travel down the quad-tree, subdividing when necessary, until a leaf node of the quad-tree is reached. If the node is

empty, the polygon will display itself and start the list of polygons for that node. If the node is populated, then the polygon will calculate its visible segments with respect to the rest of the polygons in the node, then add itself to the list of polygons. If that node is then overpopulated, the node is subdivided and the polygons divided among the sub-quadrants. Because the polygons are divided at the quadrant boundaries, the algorithm exhibits the parallelism of both the quad-tree algorithm and the geometric algorithm: all quadrants may be processed in parallel, and within each quadrant, all geometric calculations may occur simultaneously. It is important to notice that although each polygon in the priority list is processed in order, it is not necessary for one polygon to finish processing before the next begins. Thus, view the algorithm as a sequence of polygons following each other through the quad-tree, and assume the constraint that no polygon may overtake another. The quad-tree may of course grow but cannot shrink, because unlike the quad-tree algorithm, no consolidation of nodes is possible. This means that a polygon may proceed along the quad-tree without waiting for mutation of the data structure.

The primary problem with this algorithm appears to be the very existence of a population limit for a quadrant, for it is easy to conceive of scenes or views of certain scenes which might result in a number of polygons being "stacked" on top of one another. If this number is higher than the population limit, then no amount of

subdivision will resolve the issue. Special handling of such situations is necessary, and the algorithm must be smart enough to know when a situation exists which is too complicated to handle. The subdivision of quadrants must be limited to an arbitrary finite amount, possibly to the smallest resolvable unit of the display. An alternate method might be to endow the algorithm with the ability to detect polygons which completely fill a quadrant. This would seem to be a simple task, since in the process of clipping a "surrounder" polygon to a quadrant's boundaries, that clipped polygon and the quadrant would have the same vertices. Then, for all subsequent polygons which might otherwise populate that quadrant, no further processing would be necessary since the surrounder polygon has the higher priority.

The subdivision of polygons among quadrants can be done using existing methods [21], but special care is required in the interpretation of these "artificial" polygons. As shown in Figures 3-8(a-b), the polygon shown becomes four distinct polygons, each having one or more edges defined by a segment of a sub-quadrant boundary. These edges must be used in the determination of visible line segments when other polygons are clipped against them, but they must also be treated as *invisible* line segments for display purposes (else the anomalies in Figure 3-8c).

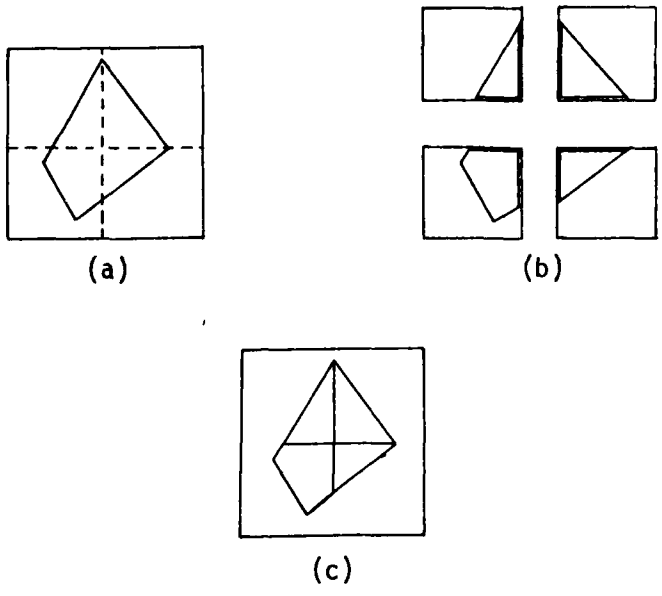


Figure 3-8

Treatment of Quadrant Boundaries as Invisible Line Segments

This approach differs from the Warnock algorithm in that the latter, for each area of the screen under consideration, processes the entire list of objects contained within that quadrant. The list of objects initially includes all objects in the scene (since the quadrant is the entire screen), and may diminish as the screen is subdivided. Determination of a surrounder polygon which hides all other polygons in a quadrant can result in early termination of processing for a paticular quadrant. The algorithm described here, however, processes the entire screen for each object, eliminating quadrants from consideration as the processing continues. The same quadrant will be

processed many times, while the Warnock algorithm processes each quadrant only once.

3.3.2 Test Results

The maximum population for a quadrant seems to be a parameter with which to tune both the amount of total work (dominated by the geometric calculations) and the amount of subdivision (dominated by the quad-tree aspects of the algorithm). For the quad-tree/geometric combination, the results from processing the test scene of 20 polygons did in fact center on the maximum allowable number of polygons in a quadrant. For an upper limit of 10 polygons, the work was roughly one-fourth that of the quad-tree method, and the critical path was essentially the same. Compared with the geometric method, though, the total work was three times as much, and the critical path was roughly ten times as long. When an upper limit of 25 polygons was set, this meant that no subdivision of the image space would be necessary, since there were only 20 polygons in the test scene. The work was thus dominated by the geometric portion of the algorithm; in fact, the total work was comparable with the purely geometric approach, but the critical path was still approximately equivalent to the quad-tree approach. This was because limitations of the simulation study implementation did not allow polygons to "follow" each other; the resultant sequential nature of the process

produced a critical path that was longer than necessary.

3.3.3 Complexity Growth Analysis

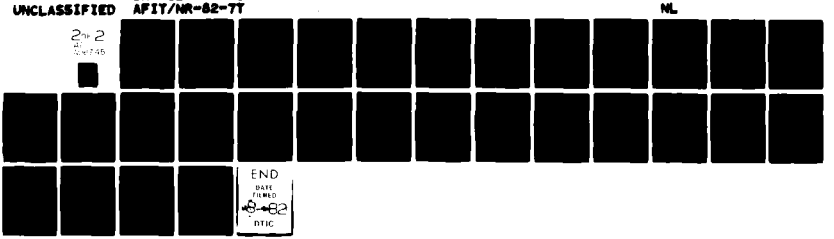
The complexity of this algorithm can be analyzed with respect to the following functions: traversal of the tree/subdivision of the polygons, and geometric calculations for hidden lines. This analysis is not a rigorous one, and is intended only for comparison with the other algorithms. The traversal and subdivision can be done in a time which is a function of the number of original polygons (n) and the population limit (p). This is O(n*log(n/p)). The geometric approach is used to process each polygon in the quadrant. Each quadrant can be processed in constant time $O(p^2)$, assuming the worst case of no available parallelism. The total processing required by the geometric portion of the algorithm is the product of this and O(n/p), the total number of nodes in the tree. The total processing thus becomes O(np). We can combine the two phases such that the total work for the algorithm is O(n*log(n/p)) + O(np), or O(n (p+log(n/p))). As *p* gets larger, the time to process the polygons with the geometric method grows as p^2, but the amount of subdivision (hence the number of nodes in the tree) decreases. This implies the existence of a *best* value for *p* such that the total execution time for the algorithm is a minimum.

Chapter 4. Summary

This thesis has dealt with the hidden surface removal problem as two distinct functions: establishment of a priority order for the polygons in a scene, and removal of the hidden lines/surfaces based on that priority order. This chapter will review the major points of object space decomposition for use in prioritization, and will summarize the three hidden surface algorithms presented in Chapter Three. Finally, some recommendations for further study will be presented, focused not on the hidden surface problem in general, but on the work done for this thesis. These will basically be descriptions of the areas the author would explore in more depth, if work on this thesis continued.

Prioritization for the polygons in a scene was made possible through decomposition of the scene's object space. The scene is enclosed in an arbitrary cubic volume and subdivided in a binary fashion until each octant of the volume is "easily interpreted." An octal-tree is used to map the resultant decomposed volume and provide a storage structure for the polygons. Prioritization for the polygons is accomplished by traversing the octal-tree in a specified order and processing the polygons as they are encountered. The traversal order is determined by the viewpoint

AD-A116 745 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH F/G 12/1
HIDDEN SURFACE REMOVAL THROUGH OBJECT SPACE DECOMPOSITION.(U)
JAN 82 R M SIMMONS
UNCLASSIFIED AFIT/NR-82-7T NL



of the scene, and is constant at all levels of the tree.

The above traversal strategy is valid only for octal-trees which contain at most one polygon per octant. This implies a requirement for physical division of the polygons at the octant boundaries, and results in a prohibitive number of artificial polygons, affecting both storage space and traversal time. A strategy of defining arbitrary "centroids" for polygons was adopted, in which decomposition occurs only until centroids are alone in their octants; no artificial polygons are created. However, the same method of traversing the octal-tree produces anomalies in the priority list, due to the fact that polygons may cross octant boundaries. This resulted in the development of a method for determining pairs of polygons which could produce anomalies in the prioritization. An algorithm was presented which tested polygons for "coupling" and associated information with those polygons for use in "decoupling" them. The traversal strategy then became one of processing uncoupled polygons as they are encountered, but requiring additional processing for any coupled polygons. The uncoupling process involves a simple comparison, and impacts on the processing time only for scenes in which the polygons are fairly completely coupled to each other.

The hidden surface algorithms presented in this thesis all assume a list of prioritized polygons as input. Each algorithm emphasizes concurrency. The first

algorithm, intended for raster scan/bit map displays, uses quad-trees to manage the display area and effect hidden surface removal. Quad-trees were defined in Section 3.1.2. Conceptually, the algorithm uses one quad-tree to represent the state of the display and one to represent a polygon. By merging the two trees such that the display tree has priority, front-to-back hidden surface removal is achieved. The main problem with this algorithm is that recursion, through binary subdivision of the display area, generally occurs down to the single pixel level. This is because the algorithm constrains quadrants to be a single color. The main work for the algorithm involves determining whether a given quadrant should be colored, uncolored, or further subdivided. In situations where recursion down to the pixel level is necessary, a great deal of unnecessary computation takes place. The algorithm exhibits a high degree of parallelism, but the computational complexity grows rapidly until the screen becomes 70% full or more (this may never happen for certain scenes).

The second algorithm deals with a geometric method for determining hidden lines. Points of intersection are calculated between the edges of each polygon and those of any polygons having higher priority. From these points, visible line segments are constructed and displayed. Virtually all computations may be done in parallel. Thus, to achieve all available parallelism, there is a requirement for $O(N^2)$ growth in the number of processors. For a small fixed number of polygons, however, the algorithm

exhibits a rapid response time.

The third and final algorithm essentially combines the first two algorithms by using quad-trees to reduce the complexity of a display so that the geometric method may be used in parallel on each small portion of the display. The prioritized polygons are apportioned to sub-quadrants of the display through binary subdivision of the screen and physical division of the polygons. Each leaf node in the quad-tree is either empty or populated with polygons (the list of polygons maintains the original priority order). Within each quadrant, the hidden lines are removed using the geometric algorithm. The number of polygons allowed in a quadrant is limited and arbitrarily small; the population limit impacts both the amount of subdivision required for the scene and the response time (because of the growth characteristics for the geometric algorithm). Each quadrant may be processed in parallel, since there is no overlap between polygons in neighboring quadrants. The algorithm exhibits a relatively small growth in complexity, being a function of the number of input polygons and the allowable quadrant population.

Further study into this area of research would concentrate on developing an operational version of the prioritizer (Chapter Two) and combination method (third algorithm in Chapter Three), implementing it on a parallel processor, and rigorous

testing with complex scenes. A producer/consumer approach would be investigated, in which the traversal of the octal-tree (with uncoupling as necessary) would produce a stream of prioritized polygons. This stream would be consumed by a process which implemented the quad-tree/geometric combination algorithm. The time required for octal-tree traversal should be proportional to the time necessary to process and display the polygons; both processes would execute simultaneously. Simulation studies for this thesis were admittedly limited in scope, and it would be interesting to see how this approach works under normal operating conditions.

References

1. Appel, A.: "The Notion of Quantitative Invisibility and the Machine Rendering of Solids," *Proc. ACM Nat. Conf.*, Thompson Books, Washiongton, D.C., 1967, p. 387.

2. Bouknight, W. J.: "A Procedure for Generation of Three-dimensional Half-toned Computer Graphics Representations," *CACM*, 13(9):527, September 1970.

3. Franklin, W. R.: Evaluation of algorithms to display vector plots on raster devices, *Computer Graphics and Image Processing*, December 1979.

4. Fuchs, H., Z. M. Kedem, and B. F. Naylor: "On Visible Surface Generation by A-Priori Tree Structures," *SIGGRAPH 80*, p. 124.

5. Galimberti, R. and U. Montanari: "An Algorithm for Hidden-Line Elimination," *CACM*,12(4):206, April 1969.

6. Griffiths, J. G.: "A Bibliography of Hidden-Line and Hidden-Surface Algorithms," *Computer Aided Design*, 10(3):203-206, May 1978.

7. Horowitz, S. L. and T. Pavlidis: "Picture segmentation by a tree traversal algorithm," *JACM*, 23(2):368, 1976.

8. Halstead, R. H.: "Architecture of a Myriaprocessor," *COMPCON 81*, San Francisco, February 1981.

9. Hunter, G. M. and K. Steiglitz: "Operations on images using quad trees," *IEEE Transactions on Pattern Recognition and Machine Intelligence*, PAM-1(2):145-153, 1979.

10. Hunter, G. M. and K. Steiglitz: "Linear Transformation of Pictures Represented by Quad Trees," *Computer Graphics and Image Processing*, 10(3):289, July 1979.

11. Klinger, A. and C. R. Dyer: "Experiments on Picture Representation Using Regular Decomposition," *Computer Graphics and Image Processing*, March 1976, p. 68.

12. Knuth, D. E.: *The Art of Computer Programming*, vol. 1: *Fundamental Algorithms*, vol. 3: *Sorting and Searching*, Addison-Wesley, Reading, Mass., 1968-1973.

13. Loutrel, P. P.: "A Solution to the Hidden-Line Problem for Computer-drawn Polyhedra," *IEEE Trans.*, ec-19(3):205, March 1970.

14. Matshushita, Y.: "A solution to the Hidden-Line Problem," *Univ. Illinois Dept. Computer Science Doc. 335*, ILLIAC IV, 1969.

15. Newell, M. E., R. G. Newell, and T. L. Sancha: "A New Approach to the Shaded Picture Problem," *Proc. ACM Nat. Conf.*, 1972.

16. Newman, W. M. and R. F. Sproull: *Principles of Interactive Computer Graphics*, 1st and 2nd ed., McGraw-Hill, New York, 1973 and 1979.

17. Reddy, D. R. and S. Rubin: "Representation of three-dimensional objects," *Carnegie-Mellon Univ., Dept. Comp. Sci.*,CMU-CS-78-113, April 1978.

18. Roberts, L. G.: "Machine Perception of Three Dimensional Solids," *MIT Lincoln Lab.*, TR 315, May 1963.

19. Romney, G. W.: "Computer Assisted Assembly and Rendering of Solids," *Univ. Utah Computer Science Dept.*, TR 4-20, 1970.

20. Sutherland, I. E., R. F. Sproull, and R. A. Schumacker: "A Characterization of Ten Hidden Surface Algorithms," *Computing Surveys*,6(1):1, March 1974.

21. Sutherland, I. E. and G. W. Hodgman: "Reentrant Polygon Clipping," CACM, 17(1):32, January 1974.

22. Warnock, J. E.: "A Hidden-Surface Algorithm for Computer Generated Half-tone Pictures," *Univ. Utah Computer Science Dept.*, TR 4-15, 1969.

23. Watkins, G. S.: "A Real-Time Visible Surface Algorithm," *Univ. Utah Computer Science Dept.*, UTEC-CSc-70-101, June 1970.

24. Weiler, L. and P. Atherton: "Hidden Surface Removal Using Polygon Area Sorting," *Computer Graphics*, 11(2):214, Summer 1977.

Appendix A

Implementation Issues

Software

The language chosen for software development was LISP, for two reasons. First, LISP provides a powerful list processing capability; the basic functionality of LISP is (CAR (CONS A B)) = A and (CDR (CONS A B)) = B. List structures are very convenient for representing graphics data structures. The language also provides a powerful programming environment for testing and debugging software, and all of the essential structured programming mechanisms (if ... then ... else, do ... while, etc.) are available in some fashion. The overriding reason for using LISP, however, was the existence of a simulator for a multi-processor system based on the MuNet parallel processor [8].

The Simulator

The simulator system consists of a front-end compiler which compiles LISP into a stack-oriented LISP-like machine language, which then runs under the simulator.

Using this approach, the hidden surface algorithms were first developed and tested using a conventional LISP interpreter and debugging environment, after which the software was transferred to the simulator in order to investigate and evaluate the parallelism inherent in each algorithm.

The compiler (LCOMP) transforms LISP into what is termed LCODE. LCOMP provides a capability for parallel processing through a basic yet powerful mechanism called "pcons" (parallel cons). In determining the value of the expression (PCONS A B), the evaluation of A and B take place in parallel. The simulator forks off two tasks which effectively execute in parallel, and the resultant values are placed into the CAR and CDR portions of a conscell. The PCONS expression may be nested, providing a virtually unlimited potential for parallelism. A considerable amount of overhead exists with the evaluation of a PCONS expression, so that careful consideration must be given to its use. As a trivial example, the evaluation of (PCONS (+ 1 2) (* 3 4)) requires three times as much processing time as does (CONS (+ 1 2) (* 3 4)), while the evaluation of complicated expressions makes the use of PCONS well worth the overhead. For instance, assuming a factorial function FACT defined in LISP as

```
(DEFUN FACT (N) (COND ((< N 2) 1) (T (* N (FACT (- N 1))))))
```

then the evaluation of (PCONS (FACT 5) (FACT 7)) results in a total of 389

instructions with a shortest path of 232, while (CONS (FACT 5) (FACT 7)) requires 371 instructions.

The existence of PCONS allows a host of auxiliary parallel constructs to be defined by the user, all of which either utilize PCONS directly, or use the same forking mechanism as PCONS. These include a parallel MAPCAR called PMAPCAR, defined as

```
(defun PMAPCAR (f l)
   (cond  ((null l) nil)
      (t (PCONS  (f (car l))
            (PMAPCAR f (cdr l))))))
```

Additionally, a PCALL construct makes use of the fact that LISP uses call-by-value for function calls; each actual parameter is evaluated in parallel by using PCONS to create the argument list. Lastly, a FORK command provides a "fork and die" capability, such that the value calculated by each sub-task is not returned to the parent process.

A Factor for Determining Parallelism

A simple way of expressing the parallelism realized from an execution of an LCODE program is by dividing the total number of instructions required of all the tasks (including overhead for forking and joining) by the number of instructions

required by the "longest path". The first figure is an expression of the amount of work performed by the processors, while the second figure gives an idea of the time required to complete the program (if all the simulated parallelism were realized). The parallelism factor is the quotient of these two figures, which gives an average parallelism (number of tasks that were active) over the course of program execution. In other words, for TI/LP = P, then if the program is to execute in time LP, an average of P tasks must be active over the course of program execution.

Systems Trivia

The systems on which development and testing of the algorithms occurred included an Interdata minicomputer and Ramtek display generator with a raster scan monitor, operating under the MagicSix operating system. The bulk of the work took place on a VAX 11/780 operating under the Unix operating system (Berkeley version), primarily for using the parallel processing simulator. Certain portions of the algorithms described have been programmed in Fortran, PL/I, C, and Lisp.

Data Structures

Three-dimensional computer graphics applications often imply the use of list and tree data structures. Modelling a 3-D scene might involve the definition of a number of

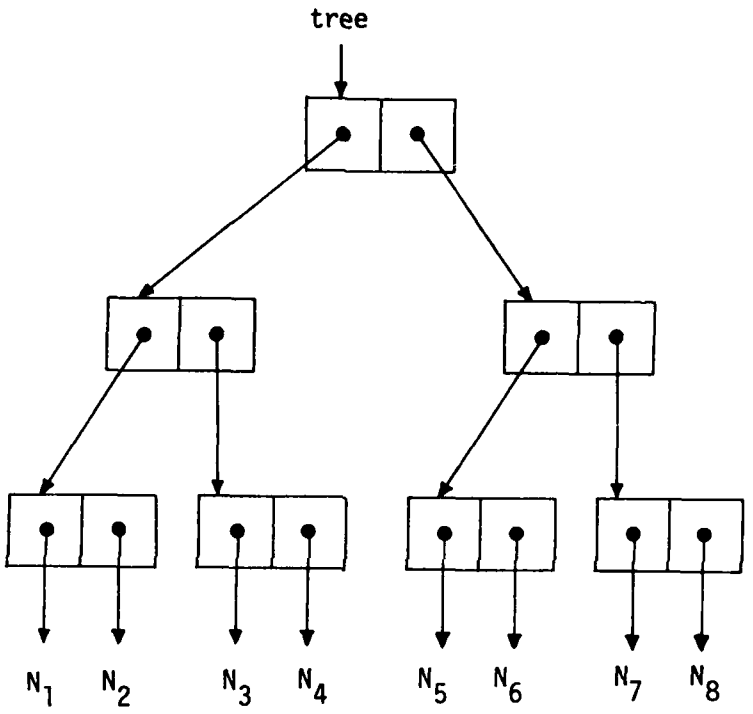
shapes, each of which is composed of an arbitrary number of polygonal surfaces. Each polygon might be described by a list of edges, and each edge would be a pair of points. A powerful list structure capability is very useful for these purposes. The tree structures, as already discussed in the case of quad- and octal-trees, are easily represented as recursively defined lists, such that each node (list) represented a subtree, and contained either four or eight subtrees, respectively. Each node of the subtree would either contain data (empty or full) or would be another subtree.

The following list structures were used in the evaluation of the hidden surface algorithm:

3D point (x y z) -- (where parentheses denote a list)
2D point (x . y) -- (the period denotes a "pair", a special case of a list)
line (point$_1$. point$_2$)
line equation (a b c) -- (ax + by + c = 0)
plane equation (a b c d) -- (ax + by + cz + d = 0)
polygon (inside_point plane_equation line$_1$ line$_2$ . . . line$_n$)
shape (center_point polygon$_1$ polygon$_2$ . . . polygon$_n$)
points list (point$_1$ point$_2$ . . . point$_n$) -- (all points in scene)
polygons list (polygon$_1$ polygon$_2$ . . . polygon$_n$) -- (all polygons in scene)
quad-tree (node$_1$ node$_2$ node$_3$ node$_4$)
node: empty (nil), full (data), quad-tree
octal-tree (((node$_1$. node$_2$) node$_3$. node$_4$) (node$_5$. node$_6$) node$_7$. node$_8$)
node: empty (nil), full (data), octal-tree

This particular representation for an octal-tree was chosen to minimize the average number of CARs and CDRs required to access any of the eight nodes in a subtree. The

box-and-pointer representation for the octal-tree is shown below.



Box-and-pointer Diagram for Octal-tree

Use of the FORK Mechanism in LCODE

There appear to be some useful applications for the FORK mechanism in LCODE, particularly in cases where the results of expression evaluations are not needed. In these graphics algorithms, the desired outcome is a number of side effects resulting in picture information (line segments and the like) being sent to a graphics

display device. In these cases, the overhead from PCONS is unnecessary since the expression values are not needed. The use of FORK in lieu of PCONS results in a significant savings in computation time. This fact was demonstrated by a short test in which the twenty polygons in the test scene were transformed from simple lists of line segments to a list which contained the known inside point, min-max boundaries of the polygon, and the line segments with the line equation added. This work was the job of the GETEQN function mentioned below. For testing PCONS, the code used was simply

```
(PMAPCAR GETEQN polgon-list)
```

and, for testing FORK,

```
(while polygon-list
 (fork (GETEQN (car polygon-list)))
 (setq polygon-list (cdr polygon-list)))
```

The results for the PCONS test were 22,585 total instructions with a longest path of 1,330 (parallelism of 17), and for the FORK test 21,866 instructions with a longest path of 707 (parallelism of 31). Part of the reason for the longer time of PCONS is the combination of PCONS overhead and function call overhead, since PMAPCAR calls

itself recursively. This test was actually not a valid use of the FORK mechanism, since in the case of using GETEQN, the results were needed in a list structure which resembled the structure of the input list of polygons, and the above code (using the FORK) would not be capable of returning such a structure.

Appendix B

Implementation of Quad-Tree Algorithm

```
function MergeTrees (old new)
    flag1 = MergeNode (old.n[1], new.n[1])
    flag2 = MergeNode (old.n[2], new.n[2])
    flag3 = MergeNode (old.n[3], new.n[3]) /* All of which can be */
    flag4 = MergeNode (old.n[4], new.n[4]) /* calculated in parallel */

    if (flag1 = flag2 = flag3 = flag4 = 'occupied') then
        return ('occupied')
    else
        return (MakeTree (flag1, flag2, flag3, flag4))
    endif
end MergeTrees


/* Get the octal tree for the new polygon */

function GetTree (polygon, x, y, size)
    if size <= 1 then
        return ('occupied')
    endif
    flag = Black-or-White? (x, y, size, polygon)
    if flag = 'black' then
        return ('occupied')
    else
    if flag = 'white' then
        return ('empty')
    endif
    mid = size / 2

/* All four invocations of GetTree may execute in parallel */

    return (MakeTree (     GetTree (polygon, x, y + mid, mid),
```

```
                GetTree (polygon, x + mid, y + mid, mid),
                GetTree (polygon, x, y, mid),
                GetTree (polygon, x + mid, y, mid)))
end GetTree


function MergeNode (old new)
    if empty? (old) then
        return (new)
    else
    if occupied? (old) then
        return (old)
    else
    if empty? (new) then
        return (old)
    else
    if occupied? (new) then
        return (new)
    else
        return (MergeTrees (old, new))
    endif
end MergeNode


/* See if any endpoints of the input lines lie inside the quadrant */

function Contains? (x, y, size, lines)
    for each line in lines do
        pt = line.p1
        if pt.x < x        then return (false) else
        if pt.x > x + size     then return (false) else
        if pt.y < y        then return (false) else
        if pt.y > y + size     then return (false) endif
    endfor
    return (true)
end Contains?


/* Check to see if quadrant should be full (black), empty (white), or */
```

```
/* subdivided (unknown). */

function Black-or-White? (x, y, size, polygon)
    p-in = polygon.inside
    pminmax = polygon.minmax
    lines = polygon.lines

/* The following four calculations may all take place in parallel */

    flag1 = Inside? (x, y + size, p-in, lines)
    flag2 = Inside? (x + size, y + size, p-in, lines)
    flag3 = Inside? (x, y, p-in, lines)
    flag4 = Inside? (x + size, y, p-in, lines)


/* Min-Max test */

    if BoxDisjoint? (x, y, size, pminmax) then return ('white') endif

/* All four corners of quadrant are inside the polygon */

    if (flag1 and flag2 and flag3 and flag4) then return ('black') endif

/* One corner of quadrant is inside the polygon */

    if (flag1 or flag2 or flag3 or flag4) then return ('unknown') endif

/* At least one polygonal vertex is inside the quadrant boundaries */

    if Contains? (x, y, size, lines) then return ('unknown') endif

/* Test to see if the quadrant boundaries and the polygon intersect anywhere */

    if x > pminmax.xmin then return (Look (lines, MakeLine (x, y, x, y + size))) endif

    if x + size < pminmax.xmax then return (Look (lines, MakeLine (x + size, y, x + size, y + size))) endif

    if y > pminmax.ymin then return (Look (lines, MakeLine (x, y, x + size, y))) endif
```

```
    if y + size < pminmax.ymax then return (Look (lines, Makeline (x, y + size, x + size, y + size))) endif

/* No intersections. . . the polygon and quadrant must be disjoint */

    return ('white')

end Black-or-White?


/* Test to see if test line intersects any of input lines */

function Look (lines, testline) returns string
    flag = 'white'
    for each line in lines do
        if Intersect (line, testline) then
            flag = 'unknown'
        endif
    endfor
    return (flag)
end Look


/* Utility function to create a line structure from two points */

function MakeLine (p1 p2) returns List
    return (CONS (EqLine (p1, p2), CONS (p1, p2)))
end MakeLine


/* For each line in the polygon, plug the two points into the line equatioin. */
/* If the signs are the same for both points w.r.t. each line, the test point */
/* is inside the polygon. */

boolean function Inside? (x-test, y-test, x-in, y-in, polygon)
    line-list = polygon.lines
    for each line in line-list do
        eq = line.eq
        a = eq.a
        b = eq.b
```

```
        c = eq.c

        val1 = a*x-test + b*y-test + c
        val2 = a*x-in + b*y-in + c

        if sign (val1) not = sign (val2) then
            return (false)
        endif
    endfor
    return (true)
end Inside?

function Merge (node, polygon, x, y, size)
    mid = s/2

    if node = 'occupied' then
        return (node)
    else
    if node = 'empty' then
        flag = Black-or-White? (polygon, x, y, size)
        if flag = 'black' then
            paint (color (polygon), x, y, size)
            return ('occupied')
        else
        if flag = 'white' then
            return ('empty')
        else
            return (MakeTree ( Merge ('empty', polygon, x, y + mid, mid),
                               Merge ('empty', polygon, x + mid, y + mid, mid),
                               Merge ('empty', polygon, x, y, mid),
                               Merge ('empty', polygon, x + mid, y, mid)))
        endif
    else
        return (MakeTree ( Merge (node[1], polygon, x, y + mid, mid),
                           Merge (node[2], polygon, x + mid, y + mid, mid),
                           Merge (node[3], polygon, x, y, mid),
                           Merge (node[4], polygon, x + mid, y, mid)))
    endif
end Merge
```

```
function Paint (color, x, y, size)  /* assumed to be intrinsic */
function MakeTree (val1, val2, val3, val4) /* assumed to be intrinsic */
```

Appendix C
Implementation of Geometric Method

```
/* This procedure takes the input prioritized list of polygons and forks */
/* off a parallel task for each polygon. The first element becomes the */
/* new polygon, the remainder of the list becomes the list of old poly- */
/* gons, and the task is started. The first element is then removed, and */
/* the process continues until the list becomes empty. */

procedure Geometric (pol-list)
    while pol-list
        new-pol = CAR (pol-list)
        old-pols = CDR (pol-list)
        FORK (ProcessPolygon (new-pol, pol-list)
        pol-list = old-pols
    endwhile
end Geometric

/* This procedure is the manager of the algorithm, being responsible for */
/* computing the invisible line segments, calculating any visible line */
/* line segments, and drawing them. */

procedure ProcessPolygon (new-pol, pol-list)

/* For each 'old' polygon, calculate the invisible line segments */

    inv-list = PMAPCAR (Compare, pol-list)

/* Create a 'new' polygon such that each line is ordered by ascending */
/* X-values, or if the X-values are equal (vertical line), then by */
/* ascending Y-values. */

    polygon = PMAPCAR (Order, new-pol)

/* Merge all the invisible line segments onto the polygon, creating */
```

```
/* a list of visible line segments for each edge in the polygon.    */

    MAPCAR (DoPol, inv-list)

    PMAPCAR (DrawPol, polygon) /* Draw the line segments which are visible */

end ProcessPolygon

/* This procedure is used to simulate a lambda expression in Lisp in  */
/* which 'new-pol' is global to this procedure. 'Compare' is used     */
/* solely as a handle for invoking the 'ComputeInvisible' procedure.  */

function Compare (old-pol) returns List
    return (ComputeInvisible (old-pol, new-pol))
end Compare


function Order (pair) return List
    p1 = CAR (pair)        /* lines are dotted pairs  */
    p2 = CDR (pair)        /* (p1 . p2)     */

    if p1.x = p2.x then       /* vertical line */
        if p1.y < p2.y then
            return (pair)     /* already in ascending Y */
        else
            return (CONS (p2, p1))  /* reverse the points */
        endif
    else              /* horizontal line */
    if p1.x < p2.x then
        return (pair)         /* already in ascending X */
    else
        return (CONS (p2, p1))  /* reverse the points */
    endif
end Order

/* This procedure is essentially a lambda mechanism for performing side  */
/* effects on the polygon line segments. 'Inv-list' is the list of      */
/* invisible line segments for the current 'old' polygon, and 'pol-segs' */
/* contains the line segments for the new polygon, already masked to     */
```

```
/* some extent by previous old polygons. Both 'inv-list' and 'pol-segs' */
/* have a number of elements equal to the number of edges in the new    */
/* polygon.                                                             */

procedure DoPol (inv-list)
    pol-segs = MAPCAR2 (ProcessMask, inv-list, pol-segs)
end DoPol

/* Nothing really special here. . . included for completeness */

procedure DrawPol (line-segs)
    MAPCAR (DrawLine, line-segs)  /* DrawLine assumed to be intrinsic */
end DrawPol

/* ComputeInvisible returns a list of invisible line segments w.r.t. the */
/* current 'old' polygon.                                                 */

function ComputeInvisible (old-pol, new-pol) returns List
    return (PMAPCAR (ProcessLine, new-pol.lines))
end ComputeInvisible

/* ProcessLine is another handle to simulate a lambda expression, used  */
/* in the above procedure.                                              */

function ProcessLine (line) returns List
    return (GetSegment (line, old-pol.p-in, old-pol.lines))
end ProcessLine

/* GetSegment does the majority of the actual work in the computation of */
/* invisible line segments. Input is the edge of the new polygon, and    */
/* the inside point and edges of the current old polygon.                */

function GetSegment (new-line, p-in, old-lines)
    p1 = new-line.p1
    p2 = new-line.p2

/* Calculate all intersection points between the new line and the old    */
/* polygon. There should be either zero, one, or two points returned.    */
/* Special handling is required in the case of the edge going through    */
```

```
/* one of the vertices of the old polygon, since one physical intersec- */
/* tion would return two identical points.    */

    intersections = GetIntersects (new-line, p-in, old-lines)

/* Check to see if either of the endpoint for the new edge are contained */
/* within the old polygon. If so, then form the invisible line segment */
/* from one or more of the endpoints and one or more of the intersection */
/* points. If neither endpoint is inside the old polygon, AND there    */
/* were no intersection points, then NULL is returned, since the line   */
/* is completely visible with respect to that polygon.    */

    if Inside? (p1.x, p1.y, p-in, old-lines) then
        if Inside? (p2.x, p2.y, p-in, old-lines) then
            return (Order (new-line)) /* Entire line invisible */
        else
            return (Order (CONS (p1, CAR (intersections))))  /* Bisect */
        endif
    else
    if (Inside? (p2.x, p2.y, p-in, old-lines) then
        return (Order (CONS (CAR (intersections), p2)))  /* Bisect */
    else
    if (intersections = NULL) then
        return (NULL)        /* Entire line visible */
    else
        return (Order (intersections)) /* Trisect */
    endif
end GetSegment

/* This procedure calculates any intersection points between the input  */
/* line and polygon (defined by 'p-in' and 'old-lines'). Since convex  */
/* polygons are assumed, there can be at most two intersection points,  */
/* so special handling is necessary to avoid any duplication.  */

function GetIntersect (new-line, p-in, old-lines)
    intersections = NULL
    for each line in old-lines do
        flag = Intersect (new-line, line) /* Get intersection point (if any) */
        if flag then
```

```
            if intersections then
                intersections = CONS (flag, intersections)
            else
                intersections = flag
            endif
        endif
    endfor
    if Length (intersections) < 3 then
        intersections = Weed (intersections) /* Obvious magic performed here */
    endif
    return (intersections)
end GetIntersects

/* ProcessMask takes an invisible line segment (the mask), and projects it */
/* onto the current list of visible line segments. This list may grow and */
/* shrink, depending on the masks which are placed on it. This function  */
/* is recursive, for use in walking down the list of segments. Notice    */
/* this routine is written for non-vertical lines only. In the case of   */
/* vertical lines, the Y-values would be compared, and a similar collec-  */
/* tion of logic would be required.       */

function ProcessMask (mask, lines) returns List
    line = CAR (lines)

    if line = NULL then return (NULL) endif  /* no more visible line segments */
    if mask = NULL then return (lines) endif /* no invisible line segment to mask */

/* Rather than attempt to explain the masking technique in English, */
/* reference is made to the diagrams in Figure 3-__ for each situa- */
/* tion which might arise.       */

/* Figure 3-__a  */

    if mask.p2.x < line.p1.x then return (lines) endif

/* Figure 3-__b  */

    if mask.p1.x > line.p2.x then
        return (CONS (line. ProcessMask (mask, CDR (lines))))
```

```
    endif

    if mask.p1.x > line.p1.x then

/*  Figure 3-__c  */

        if mask.p2.x < line.p2.x then
            return (CONS (CONS (line.p1, mask.p1),
                    CONS (CONS (mask.p2, line.p2),
                        CDR (lines))))
        else

/*  Figure 3-__d  */

            return (CONS (CONS (line.p1, mask.p1),
                    ProcessMask (mask, CDR (lines)))
        endif
    endif

/*  Figure 3-__e  */

    if mask.p2.x < line.p2.x then
        return (CONS (CONS (mask.p2, line.p2), CDR (lines)))
    endif

/*  Figure 3-__f  */

    return (CONS (line, ProcessMask (mask, CDR (lines))))
end ProcessMask
```

Appendix D

Implementation of Quad-Tree/Geometric Combination

```
procedure ProcessNode (node, polygon, x, y, size)
    if node = 'full' then
        return (node)        /* no processing needed */
    else
    if Subdivided? (node) then /* RECURSE */
        mid = size/2
        return (MakeTree ( ProcessNode (node[1], polygon, x, y + mid, mid),
                    ProcessNode (node[2], polygon, x + mid, y + mid, mid),
                    ProcessNode (node[3], polygon, x, y, mid),
                    ProcessNode (node[4], polygon, x + mid, y, mid)))
    else
    if QuadrantFull? (polygon, x, y, size) then
        return ('full')        /* Surrounder */
    else
        lines = ClipPolygon (polygon, x, y, size)
        if lines = NULL then
            return (node)  /* polygon not visible in this quadrant */
        else
        if node = 'empty' then
            DrawPolygon (lines)  /* empty quadrant; draw any part of */
            node = List (1, polygon) /* visible in quadrant and add to list */
            return (node)
        else

/* Quadrant has polygons in it; process using geometric method.  */

            ProcessPolygons (lines, node.polygons)
            node.count = node.count + 1
            node.polygons = node.polygons & polygon

/* if too many polygons are in this quadrant, divide it to reduce complexity.  */
```

```
            if node.count > MAXCOUNT then
                return (Divide (node, x, y, size))
            else
                return (node)
            endif
        endif
    endif
end ProcessNode

/* True iff each corner is inside quadrant */

function QuadrantFull? (polygon, x, y, size) returns boolean
    p-in = polygon.inside
    lines = polygon.lines

    flag1 = Inside? (x, y, p-in, lines)
    flag2 = Inside? (x, y + size, p-in, lines)
    flag3 = Inside? (x + size, y + size, p-in, lines)
    flag4 = Inside? (x + size, y, p-in, lines)

    return (flag1 and flag2 and flag3 and flag4)
end QuadrantFull?

/* Clip each line in the polygon against the window defined by x, y, and size. */

function ClipPolygon (polygon, x, y, size) returns List
    lines = polygon.lines
    new-lines = NULL

    XL = x
    XR = x + size
    YB = y
    YT = y + size

    for each line in lines do
        L = Clip (line.p1, line.p2)
        if L not = NULL then
            new-lines = new-lines & L /* '&' denotes concatenation */
        endif
```

```
    endfor

    return (new-lines)
end ClipPolygon


/* The Clip and Code routines can be found in Newmann & Sproull, */
/* Principles of Interactive Computer Graphics, 1st Edition. */


/* Get the polygon list from the node, divide quadrant into four, and for each */
/* sub-quadrant, filter out the polygons which belong. */

function Divide (node, x, y, size) returns Node
    pols = node.polygons
    mid = size/2

    return (MakeNode (    CheckPolygons (pols, x, y + mid, mid),
                CheckPolygons (pols, x + mid, y + mid, mid),
                CheckPolygons (pols, x, y, mid),
                CheckPolygons (pols, x + mid, y, mid)))
end Divide

/* Check to see if polygons belong in quadrant defined by x, y, and size. Terminate */
/* early if surrounder polygon is found. */

function CheckPolygons (polygons, x, y, size) returns List
    new-pols = NULL
    for each polygon in polygons do
        flag = Black_or_White? (polygon, x, y, size)
        if flag = 'black' then
            return ('full')
        else
        if flag = 'unknown' then
            new-pols = new-pols & polygon
        endif
    endfor
    return (new-pols)
end CheckPolygons
```